Секреты JavaScript ниндзя

Второе издание

Джон Резиг
Беэр Бибо
Иосип Марас

MANNING

Секреты JavaScript ниндзя

Второе издание

Secrets of the JavaScript Ninja

Second Edition

JOHN RESIG,
BEAR BIBEAULT,
JOSIC MARAS

MANNING

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Секреты JavaScript ниндзя

Второе издание

ББК 32.973.26-018.2.75
Р34
УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Резиг, Джон, **Бибо**, Беэр, **Марас**, Иосип.
Р34 Секреты JavaScript ниндзя, 2-е изд. : Пер. с англ. – СпБ. : ООО "Альфа-книга", 2017. – 544 с. : ил. – Парал. тит. англ.

ISBN 978-5-9908911-8-0 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Manning Publication, Co.

Authorized translation from the English language edition published by Manning Publications Co, Copyright © 2016. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2017

Научно-популярное издание

Джон Резиг, Беэр Бибо, Иосип Марас

Секреты JavaScript ниндзя

2-е издание

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 04.09.2017. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 43,86. Уч.-изд. л. 28.36.
Тираж 400 экз. Заказ № 6055.

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9908911-8-0 (рус.)
© Компьютерное издательство "Диалектика", 2017
перевод, оформление, макетирование
ISBN 978-1617-29285-9 (англ.)
© by Manning Publications Co., 2016

Оглавление

Часть I. Разминка 27

Глава 1. JavaScript повсюду 29

Глава 2. Создание страницы в динамическом режиме 41

Часть II. Представление о функциях 61

Глава 3. Функции высшего порядка для начинающих: определения и аргументы 95

Глава 4. Функции для ученика мастера: представление об их вызове 87

Глава 5. Функции для мастера: замыкания и области видимости 129

Глава 6. Функции на перспективу: генераторы и обещания 171

Часть III. Исследование объектов и упрочение кода 215

Глава 7. Объектная ориентация с помощью прототипов 217

Глава 8. Управление доступом к объектам 253

Глава 9. Работа с коллекциями 283

Глава 10. Овладение регулярными выражениями 323

Глава 11. Методики модуляризации кода 351

Часть IV. Исследование браузеров 375

Глава 12. Работа с моделью DOM 377

Глава 13. Особенности обработки событий 409

Глава 14. Стратегии разработки кросс-браузерного кода 451

Часть V. Приложения 475

Приложение А. Дополнительные средства стандарта ES6 477

Приложение Б. Средства тестирования и отладки 483

Приложение В. Ответы на упражнения 505

Предметный указатель 533

Содержание

Отзывы о первом издании 13

От автора 14

Благодарности 16

Об этой книге 18

Кому адресована книга 18
Структура книги 18
Условные обозначения, принятые в книге 21
Загружаемый исходный код 21
Как связаться с авторами 21

Об авторах 22

Об иллюстрации на обложке книги 24

От издательства 25

Часть I. Разминка 27

Глава 1. JavaScript повсюду 29

1.1. Общее представление о языке JavaScript 30
1.1.1. Дальнейшее развитие JavaScript 32
1.1.2. Транспиляторы, обеспечивающие доступ к будущему JavaScript сегодня 33
1.2. Общее представление о браузере 33
1.3. Нормы передовой практики 35
1.3.1. Отладка 36
1.3.2. Тестирование 36
1.3.3. Анализ производительности 37
1.4. Усиление переносимости приобретенных навыков 38
Резюме 39

Глава 2. Создание страницы в динамическом режиме 41

2.1. Общее представление о жизненном цикле веб-приложения 42
2.2. Стадия создания страницы 45
2.2.1. Синтаксический анализ кода HTML и построение модели DOM 46
2.2.2. Выполнение кода JavaScript 48
2.3. Обработка событий 52
2.3.1. Общее представление об обработке событий 53

Резюме 59
Упражнения 59

Часть II. Представление о функциях 61

Глава 3. Функции высшего порядка для начинающих: определения и аргументы 63

3.1. Главное отличие JavaScript как языка функционального программирования 64
3.1.1. Функции в качестве объектов высшего порядка 65
3.1.2. Функции обратного вызова 66
3.2. Особенности применения функций в качестве объектов 70
3.2.1. Сохранение функций 71
3.2.2. Самозапоминающиеся функции 73
3.3. Определение функций 75
3.3.1. Объявления функций и функциональные выражения 77
3.3.2. Стрелочные функции 82
3.4. Аргументы и параметры функций 84
3.4.1. Оставшиеся параметры 86
3.4.2. Стандартные параметры 88
Резюме 91
Упражнения 92

Глава 4. Функции для ученика мастера: представление об их вызове 95

4.1. Использование неявных параметров функции 96
4.1.1. Параметр `arguments` 96
4.1.2. Параметр `this`, представляющий контекст функции 101
4.2. Вызов функций 102
4.2.1. Вызов как функции 103
4.2.2. Вызов как метода 104
4.2.3. Вызов как конструктора 107
4.2.4. Вызов через методы `apply()` и `call()` 113
4.3. Разрешение затруднений, связанных с контекстами функций 120
4.3.1. Обращение с контекстами функций с помощью стрелочных функций 120
4.3.2. Применение метода `bind()` 124
Резюме 125
Упражнения 126

Глава 5. Функции для мастера: замыкания и области видимости 129

5.1. Общее представление о замыканиях 130
5.2. Применение замыканий на практике 134
5.2.1. Имитация закрытых переменных 134
5.2.2. Применение замыканий при обратных вызовах 136

5.3. Отслеживание выполнения кода с помощью контекстов выполнения 139
5.4. Отслеживание идентификаторов с помощью лексических сред 142
5.4.1. Вложение кода 143
5.4.2. Вложение кода и лексические среды 144
5.5. Общее представление о типах переменных в JavaScript 147
5.5.1. Изменяемость переменных 147
5.5.2. Ключевые слова для определения переменных и лексические среды 150
5.5.3. Регистрация идентификаторов в лексических средах 154
5.6. Исследование принципа действия замыканий 159
5.6.1. Еще раз об имитации закрытых переменных с помощью замыканий 159
5.6.2. Разъяснение по поводу закрытых переменных 164
5.6.3. Еще раз о замыканиях и обратных вызовах 165
Резюме 167
Упражнения 168

Глава 6. Функции на перспективу: генераторы и обещания 171

6.1. Достижение изящности асинхронного кода с помощью генераторов и обещаний 172
6.2. Использование функций-генераторов 174
6.2.1. Управление генератором с помощью итератора 176
6.2.2. Применение генераторов 179
6.2.3. Обмен данными с генератором 183
6.2.4. Исследование внутреннего механизма действия генераторов 186
6.3. Работа с обещаниями 193
6.3.1. Затруднения, связанные с простыми обратными вызовами 195
6.3.2. Углубленное исследование обещаний 197
6.3.3. Отклонение обещаний 200
6.3.4. Создание первого настоящего обещания 202
6.3.5. Связывание обещаний в цепочку 204
6.3.6. Ожидание ряда обещаний 205
6.4. Сочетание генераторов и обещаний 207
6.4.1. Асинхронные функции в перспективе 211
Резюме 212
Упражнения 213

Часть III. Исследование объектов и упрочение кода 215

Глава 7. Объектная ориентация с помощью прототипов 217

7.1. Общее представление о прототипах 218
7.2. Создание объектов и прототипы 221
7.2.1. Свойства экземпляров 224
7.2.2. Побочные эффекты динамического характера JavaScript 227

7.2.3. Типизация объектов через конструкторы 230
7.3. Достижение наследования 232
7.3.1. Трудности переопределения свойства **`constructor`** **236**
7.3.2. Операция **`instanceof`** **240**
7.4. Применение "классов" в стандарте ES6 языка JavaScript 242
7.4.1. Применение ключевого слова `class` 243
7.4.2. Реализация наследования 246
Резюме 249
Упражнения 250

Глава 8. Управление доступом к объектам 253

8.1. Управление доступом к свойствам объектов с помощью методов получения и установки 254
8.1.1. Определение методов получения и установки 256
8.1.2. Применение методов получения и установки для проверки достоверности значений свойств 262
8.1.3. Применение методов получения и установки для определения вычисляемых свойств 264
8.2. Применение прокси-объектов для управления доступом 266
8.2.1. Применение прокси-объектов для протоколирования 270
8.2.2. Применение прокси-объектов для измерения производительности 272
8.2.3. Применение прокси-объектов для автоматического заполнения свойств 274
8.2.4. Применение прокси-объектов для реализации отрицательных индексов массивов 275
8.2.5. Проблемы с производительностью при использовании прокси-объектов 278
Резюме 279
Упражнения 280

Глава 9. Работа с коллекциями 283

9.1. Массивы 284
9.1.1. Создание массивов 284
9.1.2. Добавление и удаление элементов с обоих концов массива 287
9.1.3. Добавление и удаление элементов в любом месте массива 289
9.1.4. Наиболее употребительные операции над массивами 291
9.1.5. Повторное использование встроенных методов обработки массивов 303
9.2. Отображения 305
9.2.1. Объекты непригодны в качестве отображений 306
9.2.2. Создание первого отображения 309
9.2.3. Перебор элементов отображений 313

9.3. Множества 314
9.3.1. Создание первого множества 315
9.3.2. Объединение множеств 317
9.3.3. Пересечение множеств 318
9.3.4. Разность множеств 319
Резюме 319
Упражнения 320

Глава 10. Овладение регулярными выражениями 323

10.1. Достоинства регулярных выражений 324
10.2. Основные сведения о регулярных выражениях 325
10.2.1. Назначение регулярных выражений 326
10.2.2. Члены и операции 328
10.3. Компиляция регулярных выражений 333
10.4. Фиксация совпадающих частей 336
10.4.1. Выполнение простых фиксаций 336
10.4.2. Проверка на совпадение с помощью глобальных регулярных выражений 337
10.4.3. Ссылки на фиксации 339
10.4.4. Нефиксируемые группы 340
10.5. Замена текста с помощью функций 341
10.6. Решение типичных задач с помощью регулярных выражений 344
10.6.1. Учет символов перехода на новую строку 344
10.6.2. Сопоставление с символами Юникода 346
10.6.3. Сопоставление с экранированными символами 346
Резюме 347
Упражнения 348

Глава 11. Методики модуляризации кода 351

11.1. Модуляризация кода JavaScript до появления стандарта ES6 352
11.1.1. Определение модулей с помощью объектов, замыканий и немедленно вызываемых функций 353
11.1.2. Модуляризация приложений на JavaScript по стандартам AMD и CommonJS 360
11.2. Модули по стандарту ES6 364
11.2.1. Функциональные возможности экспорта и импорта 365
Резюме 371
Упражнения 372

Часть IV. Исследование браузеров 375

Глава 12. Работа с моделью DOM 377

12.1. Вставка HTML-кода в объектную модель документа 378
12.1.1. Преобразование из формата HTML в структуру DOM 379

12.1.2. Вставка элементов разметки в документ 384
12.2. Атрибуты и свойства модели DOM 386
12.3. Трудности обращения с атрибутами стилевого оформления 388
12.3.1. Местонахождение стилей 389
12.3.2. Именование свойств стилевого оформления 391
12.3.3. Извлечение вычисленных стилей 393
12.3.4. Преобразование значений, указываемых в пикселях 397
12.3.5. Указание размеров по высоте и ширине 398
12.4. Сведение к минимуму перегрузки верстки 403
Резюме 406
Упражнения 407

Глава 13. Особенности обработки событий 409

13.1. Углубленное исследование цикла ожидания событий 410
13.1.1. Пример только с очередью макрозадач 414
13.1.2. Пример с обеими очередями макро- и микрозадач 419
13.2. Овладение таймерами: тайм-ауты и интервалы времени 424
13.2.1. Запуск обработчиков таймера в цикле ожидания событий 425
13.2.2. Преодоление трудностей затратной обработки вычислений 432
13.3. Обработка событий 436
13.3.1. Распространение событий по модели DOM 437
13.3.2. Специальные события 444
Резюме 448
Упражнения 449

Глава 14. Стратегии разработки кросс-браузерного кода 451

14.1. Соображения по поводу кросс-браузерной разработки 452
14.2. Пять самых насущных задач разработки 455
14.2.1. Программные ошибки и отличия в браузерах 456
14.2.2. Преодоление программных ошибок в браузерах 456
14.2.3. Внешний код и разметка 458
14.2.4. Регрессии 463
14.3. Стратегии реализации 465
14.3.1. Надежное устранение ошибок в кросс-браузерном коде 465
14.3.2. Обнаружение функциональных средств и полифиллы 467
14.3.3. Области непроверяемых ошибок в браузерах 469
14.4. Сокращение допущений 472
Резюме 473
Упражнения 474

Часть V. Приложения 475

Приложение А. Дополнительные средства стандарта ES6 477

Шаблонные литералы 477

Деструктурирование 479
Усовершенствованные литералы объектов 480

Приложение Б. Средства тестирования и отладки 483

Инструментальные средства для разработки и отладки веб-приложений 484
Firebug 484
Firefox Developer Tools 485
F12 Developer Tools 486
WebKit Inspector 487
Chrome DevTools 488
Отладка кода на JavaScript 488
Протоколирование 489
Точки останова 490
Заход в функцию 491
Создание тестов 494
Основы организации среды тестирования 497
Наиболее распространенные среды тестирования 499
QUnit 500
Jasmine 501
Измерение покрытия кода 502

Приложение В. Ответы на упражнения 505

Глава 2. Создание страницы в динамическом режиме 505
Глава 3. Функции высшего порядка для начинающих: определения и аргументы 506
Глава 4. Функции для ученика мастера: представление об их вызове 507
Глава 5. Функции для мастера: замыкания и области видимости 510
Глава 6. Функции на перспективу: генераторы и обещания 512
Глава 7. Объектная ориентация с помощью прототипов 514
Глава 8. Управление доступом к объектам 517
Глава 9. Работа с коллекциями 520
Глава 10. Овладение регулярными выражениями 522
Глава 11. Методики модуляризации кода 525
Глава 12. Работа с моделью DOM 527
Глава 13. Особенности обработки событий 528
Глава 14. Стратегии разработки кросс-браузерного кода 530

Предметный указатель 533

Отзывы о первом издании

"Наконец-то появилась книга, написанная мастером своего дела, где честолюбивый разработчик прикладных программ на JavaScript найдет все, что требуется знать для овладения искусством написания эффективного кросс-браузерного кода на JavaScript".

Гленн Стокол, старший ведущий методист в корпорации Oracle

"Книга написана в полном соответствии с девизом jQuery "Писать меньше, делать больше".

Андрэ Роберж, университет Св. Анны

"В книге представлены интересные и оригинальные методики".

Скотт Сойе, компания Four Winds Software

"Прочитав эту книгу, вы не будете больше слепо вставлять фрагмент кода, удивляясь, как он вообще работает. Вы будете понимать, почему он работает".

Джо Литтон, разработчик программного обеспечения для совместной работы, компания JoeLitton.net

"Эта книга поможет вам подняться во владении JavaScript до уровня мастеров".

Кристофер Хаупт, компания greenstack.com

"Вещь, которую должны знать ниндзя".

Чэд Дэвис, автор книги Struts 2 in Action

"Рекомендуется для чтения любому мастеру JavaScript".

Джон Дж. Райян III, компания Princigration LLC

"Эта книга обязательна к прочтению для любого серьезного программиста на JavaScript. Она значительно расширит ваши знания этого языка".

С., читатель из Amazon

От автора

Немыслимо, насколько переменилась среда JavaScript с тех пор, как я приступил к написанию первого издания этой книги в 2008 году. Среда, в которой мы пишем сценарии JavaScript, неузнаваемо изменилась, хотя она по-прежнему сосредоточена в основном вокруг браузера.

Популярность JavaScript в качестве полноценного, межплатформенного языка программирования заметно возросла. Теперь имеется внушительная платформа Node.js, на которой разработано бесчисленное множество приложений. Разработчики, по существу, пишут на одном языке JavaScript прикладные программы, которые можно выполнять в браузере, на сервере и даже в платформенно-ориентированном приложении на мобильном устройстве.

Но теперь еще важнее, чем когда-либо прежде, то обстоятельство, что знания разработчиков языка JavaScript должны быть на самом высоком уровне. Глубокое понимание языка и наилучших способов написания на нем кода позволяет создавать приложения, способные работать практически на любой платформе, а этим могут с полным основанием похвалиться лишь немногие языки программирования.

В отличие от предыдущих этапов развития JavaScript, теперь не наблюдается заметного роста платформенных несовместимостей. Ведь раньше приходилось ориентироваться на новые возможности самых передовых браузеров и в то же время не забывать об устаревших браузерах, по-прежнему занимавших немалую долю рынка. Мы переживаем теперь гармоничный период времени, когда большинство пользователей быстро обновляют браузеры, соревнующиеся за право называться платформой, в наибольшей степени соответствующей принятым стандартам. Производители браузеров идут даже на то, чтобы снабжать их средствами, специально предназначенными для разработчиков, надеясь тем самым упростить им жизнь.

Инструментальные средства, предоставляемые ныне браузерами и сообществом разработчиков открытого программного обеспечения, заметно опередили прежние нормы практики. Теперь у нас имеется на выбор множество сред тестирования, возможность выполнять непрерывное интеграционное тестирование, составлять отчеты о покрытии кода тестами, проводить тесты на производительность прикладных программ на реальных мобильных устрой-

ствах по всему миру и даже загружать автоматически виртуальные браузеры на любой платформе для целей тестирования.

Первое издание книги выгодно отличалось тем огромным вкладом, который в него внес Беэр Бибо, опираясь на свой немалый опыт разработчика. А настоящее издание отличается существенным вкладом, который Иосип Марас внес в исследование принципов, положенных в основу стандартов ECMAScript 6 и 7, описание норм передовой практики тестирования и представление методик, применяемых в распространенных библиотеках и каркасах JavaScript.

Все это вместе взятое существенно изменило подход к написанию кода на JavaScript. И эта книга поможет вам овладеть современными нормами передовой практики программирования на JavaScript, а также представить применяемые вами в настоящее время нормы практики разработки как единое целое, чтобы с готовностью программировать на JavaScript и в будущем.

Джон Резиг

Благодарности

Количество людей, принявших участие в работе над этой книгой, способно удивить многих. Свой плодотворный вклад в книгу, которую вы держите в руках или читаете на экране в электронном виде, внесли многие одаренные люди.

Сотрудники издательства Manning Publications неустанно трудились, чтобы добиться такого качества издания, на которое надеялись авторы книги, за что мы им искренне благодарны. Ведь без них эта книга не увидела бы свет. Мы выражаем благодарность не только издателю Маржан Бэйс (Marjan Bace) и главному редактору Дэну Махарри (Dan Maharry), но и следующим сотрудникам издательства: Озрену Харловичу (Ozren Harlovic), Грегору Зуровски (Gregor Zurowski), Кевину Салливану (Kevin Sullivan), Джанет Вэйл (Janet Vail), Тиффани Тэйлор (Tiffany Taylor), Шарон Уилки (Sharon Wilkey), Алисону Бренеру (Alyson Brener) и Гордану Салиновичу (Gordan Salinovic).

Невозможно выразить словами нашу признательность коллегам, рецензировавшим книгу и помогавшим нам довести ее до состояния полной готовности к печати: от вылавливания простых опечаток, исправления ошибок в терминах и коде и вплоть до организации книги по отдельным главам. Каждый из них внимательно просмотрел рукопись книги, прежде чем она была сдана в печать. За этот нелегкий труд мы хотели бы поблагодарить Джекоба Андерсена (Jacob Andresen), Тиджани Белмансура (Tidjani Belmansour), Франческо Бьянки (Francesco Bianchi), Мэтью Халверсона (Matthew Halverson), Беки Хьэтт (Becky Huett), Дениэла Ламба (Daniel Lamb), Михаэля Лунда (Michael Lund), Кариема Али Элкуша (Kariem Ali Elkoush), Элайз Колкер Гордон (Elyse Kolker Gordon), Кристофера Хаупта (Christopher Haupt), Майка Хэтфилда (Mike Hatfield), Герда Клевесаата (Gerd Klevesaat), Алекса Лукаса (Alex Lucas), Аруна Норонху (Arun Noronha), Адама Шеллера (Adam Scheller), Дэвида Старки (David Starkey), а также Грегора Зуровски (Gregor Zurowski).

Особая благодарность выражается Матиасу Байненсу (Mathias Bynens) и Йону Боргманну (Borgman), научным редакторам книги. Помимо проверки каждого примера кода в нескольких средах, они внесли неоценимый вклад в уточнение текста рукописи и обнаружение пропущенной информации с учетом последних изменений, внесенных для поддержки JavaScript и HTML5 в браузерах.

Джон Резиг

Мне хотелось бы поблагодарить моих родителей за их постоянную моральную и материальную поддержку в течение многих лет. Они предоставили все необходимые ресурсы и средства, чтобы пробудить во мне интерес к программированию, и с тех пор постоянно поощряли мои занятия в данной области.

Беэр Бибо

За участие в работе над этой, уже седьмой для меня книгой я хотел бы, как всегда, поблагодарить своих коллег, участников и организаторов форума coderanch.com (бывшего JavaRanch). Ведь без личного участия в этом форуме я бы никогда не начал писать книги и поэтому я искренне благодарю Пола Уитона (Paul Wheaton) и Кэти Сьерра (Kathy Sierra) за добрые напутствия, а также моральную поддержку следующих коллег, хотя и не только их одних: Эрика Паскарелло (Eric Pascarello), Эрнста Фридмана Хилла (Ernest Friedman Hill), Эндрю Монкхауза (Andrew Monkhouse), Джина Боярски (Jeanne Boyarsky), Берта Бэйтса (Bert Bates), а также Макса Хабиби (Max Habibi).

Сердечная признательность выражается моему другу Джею и моим собакам, Медвежонку и Козмо, за то, что они терпели мое присутствие и редко наделялись должным вниманием с моей стороны во время работы над этой книгой. И наконец, мне хотелось бы поблагодарить своих соавторов Джона Резига и Иосипа Мараса, без которых не состоялась бы эта книга.

Иосип Марас

Самую большую благодарность мне бы хотелось выразить моей жене Иосипе за то, что она мирилась с тем, что все свое свободное время я посвящал написанию этой книги.

Мне хотелось бы также поблагодарить Майю Стулу (Maja Stula), Дарко Стипаничева (Darko Stipanicev), Ивицу Чрнковича (Ivica Crnkovic), Яна Карлсона (Jan Carlson) и Берта Бэйтса (Bert Bates) за полезные наставления и советы, а также за проявленную снисходительность к моим "повседневным обязанностям" по мере приближения сроков сдачи книги в печать.

И, наконец, мне хотелось бы поблагодарить остальных членов моей семьи — Йере, обеих Марий, Витомира и Зденку — за то, что они всегда рядом со мной.

Об этой книге

Особое значение JavaScript было очевидно не всегда, но теперь оно несомненно. Ныне JavaScript стал одним из самых важных и широко применяемых языков программирования.

Веб-приложения служат для того, чтобы предоставить пользователям функционально богатый и удобный интерфейс, но без JavaScript в Интернете можно лишь показывать фотографии кошек. Теперь, как никогда прежде, разработчикам веб-приложений требуются прочные знания и навыки программирования на языке JavaScript, приводящем в действие эти приложения.

Но, как и завтрак с апельсиновым соком, язык JavaScript полезен не только для браузеров. Преодолев узкие границы применения в браузерах, этот язык программирования теперь применяется на серверах благодаря платформе Node.js, в настольных системах и на мобильных устройствах благодаря таким платформам, как Apache Cordova, и даже на встраиваемых устройствах благодаря платформам Espruino и Tessel.

И хотя эта книга посвящена применению сценариев JavaScript, выполняемых в браузерах, основы этого языка программирования, представленные в данной книге, выходят далеко за пределы браузеров. Ясное понимание основных понятий и овладение различными методиками и приемами программирования на JavaScript позволит читателю стать более универсальным разработчиком. С увеличением числа разработчиков, пользующихся JavaScript, теперь, как никогда прежде, стало очень важно твердо знать основы этого языка программирования, чтобы владеть им в совершенстве.

Кому адресована книга

Если вы вообще не знакомы с языком JavaScript, то это не та книга, с которой следует начинать его изучение. Тем не менее мы попытались представить в ней основные понятия JavaScript таким образом, чтобы они были ясны даже относительным новичкам. Но, откровенно говоря, данная книга больше всего подходит тем, кто уже знает JavaScript и стремится расширить свои познания этого языка, а также браузера в качестве среды, где выполняется код JavaScript.

Структура книги

Эта книга организована таким образом, чтобы вы смогли пройти весь курс обучения от ученика до мастера программирования на JavaScript. В части I

представлен основной предмет книги и подготавливается почва для изучения материала остальных ее частей.

- В главе 1 представлено введение в язык JavaScript и самые важные его средства, а также предлагаются современные нормы передовой практики, которых следует придерживаться при разработке приложений, включая тестирование и анализ производительности.
- Исследование возможностей JavaScript в этой книге проводится в контексте браузеров, и поэтому в главе 2 закладывается прочное основание введением понятия жизненного цикла веб-приложений на стороне клиента. Это поможет лучше понять роль JavaScript в процессе разработки веб-приложений.

В части II основное внимание уделяется функциям — оплоту JavaScript. В ней разъясняются причины, по которым функции так важны в JavaScript, рассматриваются разновидности функций, а также особенности определения и вызова функций. Особое внимание уделяется новому типу функций-генераторов, которые особенно удобны для обращения с асинхронным кодом.

- Глава 3 начинается с краткого экскурса в основы языка и прежде всего, как ни странно, с тщательного анализа определения *функции* в Java Script. И хотя вас в большей степени может заинтересовать *объект*, именно с твердого и ясного представления о функции, а также о том, что JavaScript является функциональным языком программирования, начинается превращение обыкновенных программирующих на JavaScript в опытных мастеров!
- Эта функциональная нить продолжается в главе 4 исследованием механизма вызова функций, а также особенностей задания неявных параметров функций.
- Не покончив еще с функциями, мы перейдем в главе 5 к более подробному изучению связанных с ними понятий *областей видимости* и *замыканий*. Замыкания, являясь основным понятием функционального программирования, позволяют точно контролировать область видимости объектов, объявляемых и создаваемых в программах. Контроль за этими областями видимости имеет решающее значение для написания грамотного кода, достойного настоящего мастера. Даже если вы перестанете читать книгу после этой главы (а мы все же надеемся, что этого не произойдет), то все равно станете намного лучше подготовленными к разработке прикладных программ на JavaScript, чем прежде.
- Исследование функций завершается в главе 6 рассмотрением совершенно нового типа функции (*генератора*) и нового типа объекта (*обещания*), помогающего оперировать асинхронными значениями. В этой главе будет также показано, как сочетать генераторы и обещания для достижения изящества при обращении с асинхронным кодом.

Часть III посвящена объектам — еще одному оплоту JavaScript. В ней тщательно исследуется объектно-ориентированный характер языка JavaScript, а также изучается порядок защиты доступа к объектам и обращения с коллекциями и регулярными выражениями.

- Объекты подробно рассматриваются в главе 7, где поясняется, каким образом в JavaScript действует не совсем обычный объектно-ориентированный механизм. В этой главе вводится также новое для JavaScript понятие классов, которые устроены глубоко внутри совсем не так, как того можно было бы ожидать.
- Исследование объектов продолжается в главе 8, где представлены различные способы защиты доступа к объектам.
- Особое внимание в главе 9 уделяется различным типам коллекций, имеющихся в JavaScript, в том числе массивам, которые существовали в JavaScript с самого начала, а также отображениям и множествам, лишь недавно внедренным в JavaScript.
- В главе 10 основное внимание уделяется регулярным выражениям — языковому средству, нередко упускаемому из виду, но позволяющему сэкономить немало строк кода при правильном употреблении. В этой главе будет показано, как составлять и употреблять регулярные выражения и как изящно решать некоторые часто встречающиеся задачи, применяя регулярные выражения и пригодные для них методы.
- В главе 11 представлены различные методики организации прикладного кода в модули — более компактные и относительно слабо связанные сегменты, улучшающие структуру и организацию прикладного кода.
- Наконец, в части IV исследуется взаимодействие JavaScript с веб-страницами и порядок обработки событий в браузере. В завершение книги будет рассмотрена важная тема разработки кросс-браузерного кода.
- В главе 12 поясняется, как динамически видоизменять страницы через интерфейсы API для манипулирования объектной моделью документа (DOM) и как оперировать атрибутами элементов разметки, свойствами и стилями. В ней также высказываются некоторые важные соображения по поводу производительности.
- В главе 13 обсуждается особое значение модели однопоточного выполнения кода JavaScript, а также влияние этой модели на цикл ожидания событий в браузере. В ней также поясняется принцип действия таймеров и временны*х* интервалов наряду с их применением для повышения воспринимаемой производительности веб-приложений.
- В заключительной главе 14 исследуются пять главных проблем разработки кросс-браузерного кода: отличия браузеров, программные ошибки и их устранение, внешний код и разметка, отсутствующие функциональные средства и регрессии. В ней подробно обсуждаются такие стратегии,

как имитация и обнаружение функциональных средств, помогающие разрешать кросс-браузерные затруднения.

Условные обозначения, принятые в книге

Весь исходный код в листингах и тексте книги набран моноширинным шрифтом, чтобы выделить его среди обычного текста. Имена функций, методов и элементов разметки в коде XML и их атрибутов также набраны моноширинным шрифтом.

В некоторых случаях исходный код переформатирован для того, чтобы он уместился на страницах книги. Как правило, исходный код написан с учетом ограничений по ширине страницы, но иногда могут встречаться незначительные отличия в его форматировании в книге по сравнению с загружаемым вариантом исходного кода. И лишь в некоторых случаях, когда длинные строки кода нельзя переформатировать, не изменяя его смысл, в листингах на страницах книги указываются метки продолжения строк кода. Многие листинги снабжены комментариями к коду, поясняющими наиболее важные понятия.

Загружаемый исходный код

Исходный код для проработки примеров, приведенных в этой книге (наряду с некоторыми дополнениями, отсутствующими в тексте книги), свободно доступен для загрузки на посвященной книге веб-странице по адресу `https://manning.com/books/secrets-of-the-javascript-ninja-second-edition`. Исходный код примеров организован в папках по отдельным главам. Такая организация специально подготовлена для размещения на локальном веб-сервере, например Apache HTTP Server. Для этого достаточно извлечь загруженный код из архива в избранную папку и сделать ее корневой для документов веб-сервера.

За некоторыми исключениями, для большинства примеров наличие веб-сервера вообще не требуется. А при желании их исходный код можно загрузить на выполнение непосредственно в браузер.

Как связаться с авторами

Авторы и издательство Manning Publications приглашают читателей на форум, посвященный этой книге, где они могут разместить свои комментарии к книге, задать вопросы технического характера и получить помощь от авторов и других пользователей JavaScript. Для доступа и подписки на форум введите в окне своего браузера адрес `https://manning.com/books/secrets-of-the-javascript-ninja-second-edition` и щелкните на ссылке `Book Forum` (Форум для обсуждения книги). На открывшейся странице появятся сведения о том, как войти на форум после регистрации, о видах доступной помощи, а также о правилах ведения форума.

Об авторах

Джон Резиг является штатным инженером в Академии Хана и создателем библиотеки jQuery для JavaScript. Помимо первого издания данной книги, он является также одним из авторов книги *ProJavaScript Techniques* (в русском переводе вышла под названием *JavaScript для профессионалов* в ИД "Вильямс"; 2016 г.).

Джон разработал всеобъемлющую базу данных, а также механизм поиска изображений укиё-э — японской гравюры на дереве, доступный по адресу `https://ukiyo-e.org/`. Он входит в совет Американского общества любителей японского искусства и является внештатным исследователем в японском университете Ритсумейкан, где он изучает искусство укиё-э. Проживает он в нью-йоркском районе Бруклин.

Беэр Бибо занимается программированием более трех десятилетий, начиная с программы, написанной для игры в крестики-нолики на супер ЭВМ Control Data Cyber через телетайп на скорости 100 бод. Имея два диплома инженера-электрика, Беэр занимался разработкой антенн и аналогичной аппаратуры, но, начиная с первой работы в корпорации Digital Equipment Corporation, его всегда больше увлекало программирование.

Беэр выполнял проекты на заказ для таких компаний, как Dragon Systems, Works.com, Spredfast, Logitech, Caringo и многих других коммерческих организаций. Он даже служил в армии США, обучая солдат-пехотинцев боевому искусству подрывать танки, что очень пригодилось ему впоследствии на совещаниях разработчиков программного обеспечения, где в обстановке бурных дискуссий обсуждались насущные задачи текущих проектов. В настоящее время Беэр работает разработчиком внешнего интерфейса в ведущей компании, поставляющей программное обеспечение для хранения объектов с возможностями обширного масштабирования и защиты содержимого.

Помимо первого издания данной книги, Беэр является автором целого ряда других книг, вышедших в издательстве Manning Publications: *jQuery in Action*

(первое и второе издания), *Ajax in Practice* (*Ajax на практике*, пер. с англ., ИД "Вильямс", 2008) и *Prototype and Scriptaculous in Action* (*AJAX: библиотеки Prototype и Scriptaculous в действии*, пер. с англ., ИД "Вильямс", 2008). Кроме того, он был научным рецензентом многих книг по разработке веб-приложений из серии *Head First*, вышедших в издательстве O'Reilly Publishing: *Head First Ajax*, *Head Rush Ajax* и *Head First Servlets and JSP*.

В свободное от работы за компьютером время Беэр любит готовить сытные обеды, о чем свидетельствует размер его брюк, снимать на фото- и видеокамеру, ездить на своем мотоцикле марки Yamaha V-Star и носить рубашки с принтом в тропическом стиле. Он работает и проживает в городе Остин, шт. Техас, который очень любит, за исключением совершенно бесшабашных водителей.

Иосип Марас является постдокторантом, занимающимся научными исследованиями на факультете электротехники, механики и кораблестроения в университете города Сплит, Хорватия. Он получил докторскую степень в области программотехники, защитив диссертацию на тему "Автоматизация повторного использования кода в разработке веб-приложений", которая, среди прочего, содержала реализацию интерпретатора JavaScript на самом языке JavaScript. В ходе своих исследований он опубликовал более десятка трудов и докладов в научных изданиях и на конференциях, посвященных, главным образом, программному анализу клиентских веб-приложений.

Помимо научных занятий, Иосип преподает веб-разработку, системный анализ и проектирование, а также разработку приложений для Windows (за последние шесть лет он выпустил около двух сотен студентов). Он также владеет небольшой компанией, разрабатывающей программное обеспечение. Свое личное время Иосип предпочитает отдавать чтению, длительным прогулкам, если позволяет погода, а также плаванию в Адриатическом море.

Об иллюстрации на обложке книги

Фигура воина на обложке книги взята с гравюры на дереве "Актер *но* в роли самурая" неизвестного японского художника середины XIX века. *Но* — это форма музыкальной драмы в классическом японском театре, которую исполняют начиная с XIV века. Многие персонажи подобных драм играют в масках, причем мужские и женские роли исполняют только мужчины. Самурай — это героическая фигура японской истории, нередко представленная в театре и изобразительном искусстве с большим мастерством, пышностью наряда и суровостью настоящего воина.

Самураи и ниндзя были воинами, отличавшимися в японском военном искусстве своим мастерством, смелостью и хитростью. Самураи относились к военной элите, были хорошо образованы, умели читать и писать, а также искусно воевать. Они были связаны строгим кодексом чести под названием бусидо, что означает путь воина. Этот кодекс чести передавался изустно из поколения в поколение, начиная с X века. Аналогично европейским рыцарям, самураев набирали из аристократов и высших слоев японского общества. Они шли на битву в строгом боевом порядке в сложных доспехах и ярких одеждах, чтобы произвести впечатление и устрашить противника. А ниндзя отличались в большей степени своим боевым искусством, чем общественным положением или образованностью. Одевались они в черное, закрывая свое лицо, выполняли боевые задания в одиночку или небольшими группами, нападая на противника скрытно, с хитрыми уловками и применяя любую тактику для достижения успеха. Единственным правилом их поведения была скрытность.

Иллюстрация на обложке книги отобрана с трех японских гравюр, которые многие годы принадлежат главному редактору издательства Manning Publications. И когда авторы искали походящее изображение ниндзя для обложки этой книги, их внимание привлекла поразительная гравюра самурая, которую они отобрали в качестве иллюстрации благодаря ее сложной детализации, ярким краскам и выразительному изображению сурового воина, готового нанести удар и победить.

В наше время, когда одну компьютерную книгу трудно отличить от другой, издательство Manning Publications славится своей изобретательностью и инициативностью в оформлении обложек книг, издаваемых более двадцати лет, где наглядно проявляется все многообразие мирового изобразительного искусства. И характерным тому примером служит обложка этой книги.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: **info@dialektika.com**

WWW: **http://www.dialektika.com**

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116

в Украине: 03150, Киев, а/я 152

Часть I

Разминка

В этой части подготавливается почва для обучения искусству программирования на JavaScript. В главе 1 рассматривается текущее состояние языка JavaScript и исследуются некоторые среды, в которых может выполняться код JavaScript. Особое внимание в этой главе будет уделено *браузеру* — среде, с которой все начинается. Попутно будет представлен ряд норм передовой практики, применяемых при разработке приложений на JavaScript.

В связи с тем что исследование JavaScript будет проведено в контексте браузеров, в главе 2 вам предстоит изучить жизненный цикл клиентских веб-приложений и ознакомиться с порядком выполнения кода JavaScript, вписывающегося в этот жизненный цикл. Завершив чтение этой части книги, вы будете готовы приступить к обучению искусству программирования на JavaScript.

1 JavaScript повсюду

В этой главе...

- Основные языковые средства JavaScript
- Основные элементы интерпретатора JavaScript
- Три нормы передовой практики разработки приложений на JavaScript

В качестве примера рассмотрим профессиональную карьеру некоего Боба. Изучая в течение нескольких лет создание настольных приложений на C++, он был выпущен в широкий мир дипломированным разработчиком программного обеспечения в начале 2000-х годов. В то время веб только получила широкое распространение, и всем хотелось повторить успех компании Amazon. Поэтому прежде всего он обратился к изучению веб-разработки.

Боб изучил язык PHP, чтобы динамически формировать веб-страницы, на которые он обычно помещал сценарии JavaScript для достижения сложных функциональных возможностей вроде проверки достоверности введенных данных формы и даже отсчета времени на странице в динамическом режиме! Несколько лет спустя появились смартфоны, и, предвосхищая появление нового крупного рынка, Боб предусмотрительно изучил языки Objective-C и Java, чтобы разрабатывать приложения для мобильных устройств, работающие под управлением iOS и Android.

В течение своей профессиональной карьеры Боб создал немало успешных приложений, допускавших расширение и сопровождение. Но, к сожалению, постоянно сменяя все эти языки программирования и каркасы приложений, бедный Боб стал выбиваться из сил.

А теперь рассмотрим профессиональную карьеру некоей Анны. Она получила образование в области разработки программного обеспечения, специализируясь на облачных и веб-приложениях. Она создала ряд веб-приложений среднего масштаба на основе современных каркасов MVC (модель–представление–контроллер) наряду с приложениями для мобильных устройств, работающими под управлением iOS и Android. Кроме того, она создала настольное приложение, работающее под управлением Linux, Windows, Mac OS X, и даже приступила к построению безсерверной версии этого приложения, полностью размещаемой в облаке. *И все, что она сделала, было написана на JavaScript.*

Подумать только! То, на что Бобу потребовалось 10 лет и 5 языков, Анне удалось достичь за 2 года только на *одном* языке. И это крайне редкий случай в истории вычислительной техники, когда конкретные знания настолько легко и удобно применяются в столь разных предметных областях.

Зародившись из скромного 10-дневного проекта в 1995 году, язык программирования JavaScript теперь стал одним из самых распространенных в мире. Он применяется буквально *повсюду* благодаря более эффективным механизмам и внедрению таких каркасов и платформ, как Node, Apache Cordova, Ionic и Electron, которые расширяют возможности JavaScript далеко за пределы скромной веб-страницы. И подобно HTML, сам язык JavaScript претерпевает ныне существенные, хотя и запоздалые обновления, чтобы стать еще более пригодным для разработки современных приложений.

В этой книге мы собираемся ознакомить вас со всеми особенностями языка JavaScript. Относите ли вы себя к числу таких разработчиков, как Боб или Анна, проработав материал этой книги, вы сможете разрабатывать самые разные приложения как в новых, так и в действующих проектах.

Знаете ли вы?

Что такое Babel и Traceur и почему эти средства так важны для разработки современных приложений на JavaScript?

Какие основные части интерфейса JavaScript API любого браузера применяются в веб-приложениях?

1.1. Общее представление о языке JavaScript

Многие программирующие на JavaScript подобно Бобу и Анне в течение своей профессиональной карьеры достигают того момента, когда они активно пользуются обширным рядом элементов, образующих данный язык. Но зачастую эти навыки не выходят за пределы элементарных уровней. На наш взгляд, это происходит потому, что JavaScript, где применяется C-подобный синтаксис, отдаленно напоминает другие широко распространенные C-подобные языки вроде C# и Java, оставляя впечатление сходства с ними.

Программисты нередко считают, что если они знают C# или Java, то уже в достаточной степени разбираются в принципе действия JavaScript. Но это глубокое заблуждение! В сравнении с другими широко распространенными

языками программирования, язык JavaScript намного более *функционально* ориентированный. Некоторые понятия JavaScript коренным образом отличаются от аналогичных понятий в других языках программирования.

К числу подобных отличий относятся следующие.

- **Функции являются объектами высшего порядка.** Функции в JavaScript сосуществуют с любым другим объектом JavaScript и могут трактоваться как объекты. Их можно создавать с помощью литералов, обращаться к ним с помощью переменных, передавать их в качестве аргументов функций и даже возвращать в качестве значений, возвращаемых функциями. Большая часть главы 3 будет посвящена исследованию тех замечательных преимуществ, которые функции в качестве объектов высшего порядка приносят в коде JavaScript.
- **Замыкания функций.** Понятие замыкания функции обычно понимается плохо, но в то же время оно основательно и безоговорочно подчеркивает особое значение функций в JavaScript. Многие преимущества замыканий будут раскрыты в главе 5, а до тех пор достаточно знать, что функция является *замыканием*, если она активно поддерживает (т.е. "замыкает") внешние переменные, применяемые в ее теле. Помимо замыканий, мы подробно рассмотрим многие особенности самих функций в главах 3 и 4, а области видимости идентификаторов – в главе 5.
- **Области видимости.** До недавнего времени в JavaScript отсутствовали переменные с блочной областью видимости, как в других C-подобных языках. Вместо этого приходилось использовать только на глобальные переменные, а также переменные, действующие на уровне функций.
- **Объектная ориентация на основе прототипов.** В отличие от других широко распространенных языков программирования, в том числе C#, Java и Ruby, в JavaScript применяются прототипы. Зачастую разработчики, переходящие на JavaScript из языков, основанных на классах (например, Java), пытаются применять JavaScript, как будто это Java, особенно при написании опирающегося на классы кода Java с использованием синтаксиса JavaScript. И тогда они с удивлением обнаруживают, что получаемые результаты отличаются от ожидаемых. Именно поэтому мы подробно рассмотрим в части III данной книги прототипы, принцип действия объектной ориентации на основе прототипов и особенности ее реализации в JavaScript.

В основу JavaScript положена тесная взаимосвязь между прототипами и объектами, функциями и замыканиями. Ясное представление о сильной взаимосвязи между этими понятиями позволяет заметно усовершенствовать навыки программирования на JavaScript, закладывая основы для разработки приложений любого типа, независимо от того, где выполняется код JavaScript: на вебстранице, в настольном приложении, в приложении для мобильного устройства или на сервере.

Помимо перечисленных выше основополагающих понятий, имеются другие языковые средства JavaScript, помогающие в написании более изящного и эффективного кода. Некоторые из этих средств знакомы таким опытным разработчикам, как Боб, из других языков программирования вроде Java и C++. В частности, мы рассмотрим следующие языковые средства JavaScript.

- **Генераторы.** Это функции, способные генерировать несколько значений по запросам и приостанавливать свое выполнение в промежутках между запросами.
- **Обещания.** Обеспечивают более полный контроль над выполнением асинхронного кода.
- **Прокси-объекты.** Позволяют управлять доступом к некоторым объектам.
- **Усовершенствованные методы обработки массивов.** Позволяют сделать намного более изящным код обработки массивов.
- **Отображения и множества.** Служат для создания словарных коллекций, а также коллекций однозначных элементов.
- **Регулярные выражения.** Позволяют упростить фрагменты кода, которые иначе получаются довольно сложными.
- **Модули.** Служат для разделения кода на более мелкие, относительно самостоятельные части, упрощающие ведение крупных проектов.

Глубокое понимание основ и умение пользоваться дополнительными языковыми средствами с наибольшей выгодой позволяют поднять качество прикладного кода на более высокий уровень. Оттачивая свое мастерство с целью связать все перечисленные выше понятия и языковые средства вместе, вы достигнете такого уровня понимания, на котором сможете создавать любой доступный вам тип приложения на JavaScript.

1.1.1. Дальнейшее развитие JavaScript

Комитет ECMAScript, отвечающий за стандартизацию JavaScript, недавно завершил работу над стандартом ES7/ES2016 языка JavaScript. В стандарте ES7 произведена незначительная модернизация JavaScript (по крайней мере, в сравнении со стандартом ES6), поскольку комитет ECMAScript преследовал цель уделить основное внимание внесению мелких ежегодных изменений в язык.

В данной книге подробно исследуется стандарт ES6, но в то же время уделяется внимание таким языковым средствам, появившимся в стандарте ES7, как новая функция `async()`, помогающая в обращении с асинхронным кодом, как поясняется в главе 6.

Ежегодные постепенные обновления спецификации языка явно свидетельствуют об его успешном развитии, но это совсем не означает, что новые языковые средства станут доступными разработчикам веб-приложений, как только будет выпущена спецификация. Код JavaScript должен выполняться интер-

претатором (или движком) JavaScript, и поэтому мы нередко с нетерпением ожидаем обновлений избранных нами интерпретаторов JavaScript, в которые включены эти новые привлекательные языковые средства.

На заметку

При рассмотрении языковых средств JavaScript, определенных в стандарте ES6/ES2015 или ES7/ES2016, на полях данной книги будут встречаться приведенные здесь пиктограммы наряду со ссылками на сведения о поддержке соответствующих версий JavaScript в браузерах.

К сожалению, всегда существует вероятность, что новые языковые средства, которыми так хотелось бы воспользоваться, еще не поддерживаются в конкретном браузере. Хотя разработчики интерпретатора JavaScript стараются поспеть вовремя с выпуском очередного стандарта и сделать для этого все, что в их силах. Правда, текущее состояние поддержки языковых средств JavaScript в различных браузерах можно отслеживать в списках, доступных по следующим адресам: `https://kangax.github.io/compat-table/es6/`, `http://kangax.github.io/compat-table/es2016plus/` и `https://kangax.github.io/compat-table/esnext/`.

1.1.2. Транспиляторы, обеспечивающие доступ к будущему JavaScript сегодня

В связи с частыми циклами выпуска браузеров нам обычно не приходится долго ждать появления в них поддержки какого-нибудь языкового средства JavaScript. А что, если нам требуется воспользоваться всеми преимуществами новейших языковых средств JavaScript, но мы пребываем в плену жестокой реальности, когда пользователи наших веб-приложений по-прежнему имеют устаревшие браузеры?

В качестве выхода из столь затруднительного положения можно воспользоваться *транспиляторами* (от сочетания слов "трансформация + компиляция") — инструментальными средствами, преобразующими самый передовой код JavaScript в эквивалентный (а если это невозможно, то в аналогичный) код, надлежащим образом действующий в большинстве браузеров.

В этой книге особое внимание уделяется выполнению кода JavaScript в браузере. Чтобы эффективно пользоваться браузерной платформой, придется основательно изучить внутренний механизм работы браузеров. Итак, приступим!

1.2. Общее представление о браузере

Ныне приложения на JavaScript могут выполняться во многих средах. Но мы, как правило, начинаем с *браузера* — среды, из которой происходят все остальные среды и на которой мы сосредоточим основное внимание. Именно

браузер предоставляет различные понятия и прикладные программные интерфейсы (API) для тщательного исследования (рис. 1.1).

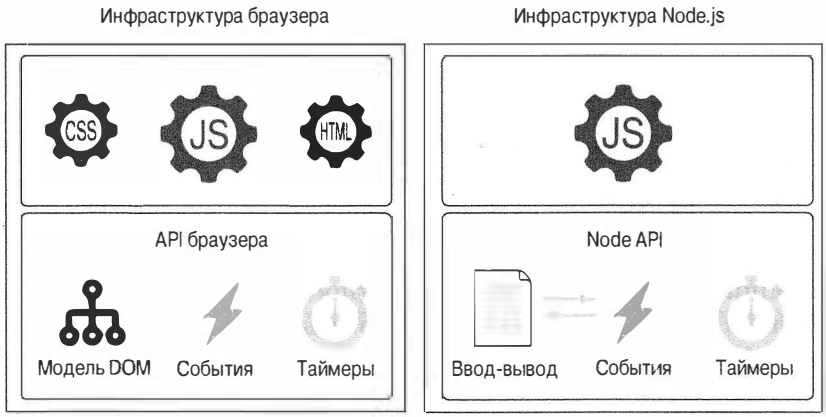


Рис. 1.1. Клиентские веб-приложения опираются на инфраструктуру, предоставляемую браузером. Особое внимание мы уделим объектной модели документа (DOM), событиям, таймерам и API браузера

В частности, мы сосредоточим основное внимание на следующем.

- **Объектная модель документа (DOM).** Модель DOM представляет собой структурированное представление пользовательского интерфейса клиентского приложения, которое, хотя бы первоначально, строится на основе HTML-кода веб-приложения. Чтобы разрабатывать удачные приложения, необходимо иметь не только глубокое понимание внутреннего механизма работы JavaScript, но и научиться строить модель DOM (см. главу 2) и писать эффективный код, манипулирующий этой моделью (см. главу 12). Благодаря этому в ваших руках окажется создание передовых, высокодинамичных пользовательских интерфейсов.
- **События.** Подавляющее большинство приложений на JavaScript являются *управляемыми событиями*. Это означает, что большая часть прикладного кода выполняется в контексте реагирования на конкретное событие. Характерные тому примеры – события, наступающие в сети, при срабатывании таймеров, в результате действий пользователя (например, события от щелчков кнопками или перемещения мыши, нажатия клавиш и т.д.). Именно поэтому мы подробно рассмотрим внутренние механизмы событий в главе 13, уделив особое внимание *таймерам*, которые нередко пребывают втайне, но позволяют решать такие сложные задачи программирования, как, например, длительные вычисления и плавная анимация.
- **API браузера.** Чтобы упростить взаимодействие с внешним миром, в браузере предоставляется прикладной программный интерфейс (API), который позволяет получать информацию об устройствах, сохранять

данные локально или связываться с удаленными серверами. Некоторые интерфейсы API будут исследованы в данной книге.

Совершенствуя свои навыки программирования на JavaScript и углубляя свои знания интерфейсов API, предоставляемых браузерами, вы сможете продвинуться далеко вперед. Но рано или поздно вам придется столкнуться с различными трудностями применения и вопросами несовместимости *браузеров*. В идеальном случае все браузеры должны работать безошибочно и поддерживать на постоянной основе веб-стандарты, но всем нам хорошо известно, что в действительности дело обстоит совсем иначе.

Качество браузеров за последнее время значительно улучшилось, тем не менее им по-прежнему присущи программные ошибки, отсутствие необходимых интерфейсов API и характерные особенности работы, которые приходится учитывать при разработке веб-приложений. Выработать комплексную стратегию разрешения затруднений, связанных с браузерами, и хорошо знать их отличия и особенности работы не менее важно, если не важнее, чем грамотно программировать на JavaScript.

При написании браузерных приложений или применяемых в них библиотек JavaScript очень важно правильно выбрать поддержку конкретных браузеров. Разумеется, хотелось бы поддерживать все имеющиеся браузеры, но ограничения, накладываемые на ресурсы разработки и тестирования, диктуют совершенно иной подход. Именно поэтому мы тщательно исследуем стратегии разработки кросс-браузерного кода в главе 14.

Стоимость затрат на разработку кросс-браузерного кода может в значительной степени зависеть от квалификации и опыта самих разработчиков. И эта книга призвана повысить уровень квалификации тех читателей, которые в этом кровно заинтересованы, поэтому перейдем к рассмотрению современных норм передовой практики.

1.3. Нормы передовой практики

Безусловно, иметь хорошие навыки программирования на JavaScript и солидный опыт авторской разработки кросс-браузерного кода никогда не помешает, но это еще не все. Чтобы стать по-настоящему квалифицированным разработчиком приложений на JavaScript, необходимо также освоить наилучшие образцы, приемы и способы, накопленные в программировании предыдущими поколениями разработчиков для написания качественного кода. Эти образцы, приемы и способы называются *нормами передовой практики* и, помимо совершенного владения языком программирования, включают в себя следующие элементы.

- Отладка.
- Тестирование.
- Анализ производительности.

При написании кода очень важно придерживаться этих норм практики, что и будет сделано на протяжении всей данной книги. Поэтому рассмотрим далее некоторые из них.

1.3.1. Отладка

Отладка кода JavaScript раньше означала обращение к функции `alert()` для проверки значений переменных. Правда, за последние годы возможности отладки кода JavaScript значительно расширились и не в последнюю очередь благодаря широкому распространению расширения Firebug браузера Firefox для веб-разработки. Аналогичные средства веб-разработки имеются теперь и в других основных браузерах.

- **Firebug** – широко распространенное расширение браузера Firefox для разработчиков (`http://getfirebug.com/`).
- **Chrome DevTools** – инструментальное средство, разработанное создателями браузера Chrome и применяемое в браузерах Chrome и Opera.
- **Firefox Developer Tools** – инструментальное средство, разработанное создателями браузера Firefox и входящее в его состав.
- **F12 Developer Tools** – инструментальное средство, входящее в состав браузеров Internet Explorer и Microsoft Edge.
- **WebKit Inspector** – инструментальное средство, применяемое в браузере Safari.

Как видите, в каждом из основных браузеров специально предоставляются инструментальные средства, которыми можно пользоваться для отладки веб-приложений. А времена, когда для отладки кода JavaScript применялась функция `alert()`, давно канули в Лету!

Все эти инструментальные средства построены на сходных принципах, которые были внедрены, главным образом, в Firebug, и поэтому они предоставляют сходные функциональные возможности, в том числе исследование модели DOM, отладка кода JavaScript, редактирование стилей CSS, обработка сетевых событий и т.д. Любое из перечисленных выше инструментальных средств вполне справляется с задачами отладки, поэтому пользуйтесь тем, которое имеется в избранном вами браузере или же в том браузере, где требуется выявить программные ошибки.

Кроме того, некоторыми из инструментальных средств (например, Chrome Dev Tools) можно воспользоваться для отладки других типов приложений, в том числе приложений на платформе Node.js. (Некоторые методики отладки будут представлены в приложении Б к данной книге.)

1.3.2. Тестирование

В примерах, приведенных в данной книге, мы будем активно пользоваться целым рядом методик тестирования, которые послужат не только для провер-

ки правильности кода, выполняемого в этих примерах, но и в качестве образца для тестирования кода вообще. Основным инструментальным средством, которым мы будем пользоваться для тестирования кода, служит функция `assert()`, назначение которой — утвердить, что исходная предпосылка истинна или ложна. Определяя утверждения, мы можем проверить, ведет ли себя код именно так, как и предполагалось. Ниже приведена общая форма данной функции.

```
assert(условие, сообщение);
```

В качестве первого параметра указывается `условие`, которое должно быть истинно, а в качестве второго параметра — `сообщение`, выводимое в противном случае.

Рассмотрим следующий пример:

```
assert(a == 1, "Проблема! a не равно 1!");
```

Если значение переменной `a` не равно **1**, утверждение не выполняется и выводится сообщение, предупреждающее об этом.

Примечание

Следует иметь в виду, что функция **`assert()`** не относится к стандартным языковым средствам JavaScript. Поэтому нам придется самостоятельно реализовать эту функцию в виде метода в приложении Б.

1.3.3. Анализ производительности

Другим не менее важным практическим приемом является анализ производительности. Интерпретаторы JavaScript, действующие в браузерах, претерпели значительные усовершенствования с точки зрения производительности самого языка JavaScript, но это совсем не означает, что можно писать небрежный и неэффективный код.

Приведенным ниже фрагментом кода мы будем пользоваться далее в этой книге для сбора данных о производительности.

```
console.time("My operation");          <-- Запустить таймер

for(var n = 0; n < maxCount; n++){     |
/* выполнить измеряемую операцию */    | Многократно выполнить операцию
}                                      |

console.timeEnd("My operation");       <-- Остановить таймер
```

В приведенном выше фрагменте исполнение измеряемого на производительность кода заключается в вилку между вызовами методов `time()` и `timeEnd()` встроенного объекта `console`.

Перед началом операции делается вызов `console.time()` для запуска таймера по имени (в данном случае — `"My operation"`). Затем код выполняется в цикле `for` определенное количество раз (в данном случае это количество определяется значением переменной `maxCount`). Точно измерить время одно-

кратного выполнения кода очень трудно, поскольку оно происходит очень быстро, поэтому для получения измеримых величин код приходится выполнять многократно. Зачастую количество повторений кода исчисляется десятками, сотнями тысяч и даже миллионами в зависимости от характера измеряемого на производительность кода. Подходящее значение переменной `maxCount` можно подобрать методом проб и ошибок.

После завершения операции вызывается метод `console.timeEnd()`, которому передается такое же имя таймера. В результате на консоли браузера появится значение интервала времени, которое прошло с момента запуска таймера.

Рассмотренные выше и другие нормы передовой практики, которые будут представлены далее в книге, в значительной мере способствуют усовершенствованию разработки веб-приложений на JavaScript. Для разработки подобных приложений с ограниченными ресурсами, предоставляемыми браузером, необходимы прочные и полные знания и навыки программирования, особенно если учесть постоянное усложнение функциональных возможностей и совместимости браузеров.

1.4. Усиление переносимости приобретенных навыков

Когда Боб только начинал изучать веб-разработку, в каждом браузере посвоему интерпретировались сценарии и стили пользовательского интерфейса, причем создатели браузеров пропагандировали свой способ интерпретации как самый лучший, что вынуждало веб-разработчиков скрежетать зубами от отчаяния. Правда, войны браузеров окончились, как только были стандартизированы HTML, CSS, интерфейс DOM API и JavaScript, а разработчики смогли уделить основное внимание эффективности кросс-браузерных приложений на JavaScript. Безусловно, такой подход к веб-сайтам как к веб-приложениям привел к появлению новых идей, инструментальных средств и методик для перехода от настольных приложений к веб-приложениям. И теперь такой перенос знаний, методик и инструментальных средств, возникших в разработке клиентских веб-приложений, проник в другие прикладные области.

Таким образом, стремясь глубоко познать основные принципы действия JavaScript и базовых интерфейсов API, можно стать более универсальным разработчиком. Применяя браузеры и Node.js (производную от браузера среду), можно разработать практически любой тип приложения, в том числе следующие.

- **Настольные приложения**, используя, например, NW.js (`http://nwjs.io/`) или Electron (`http://electron.atom.io/`). Эти технологии обычно включают в себя браузер, что позволяет создавать пользовательские интерфейсы настольных приложений стандартными средствами HTML, CSS и JavaScript, опираясь на базовые знания JavaScript и браузеров, а также дополнительную поддержку взаимодействия с файловой системой.

И это дает возможность строить по-настоящему платформенно-независимые настольные приложения, которые выглядят одинаково в Windows, Mac OS X и Linux.

- **Приложения для мобильных устройств**, используя такие каркасы, как Apache Cordova (`https://cordova.apache.org/`). Как и при построении настольных приложений с помощью веб-технологий, в каркасах приложений для мобильных устройств применяется встроенный браузер, но с дополнительными интерфейсами API для взаимодействия с конкретными мобильными платформами.
- **Серверные приложения и приложения для встраиваемых устройств**, используя Node.js – производную от браузера среду, где применяются многие из тех базовых принципов, которые служат для построения браузеров. Например, код JavaScript выполняется в среде Node.js на основании событий.

Анна даже не знает, насколько ей повезло, тогда как Боб хорошо это понимает. И совершенно не важно, требуется ли ей написать стандартное настольное приложение, приложение для мобильного устройства, сервера или даже встраиваемого устройства – все эти типы приложений разделяют ряд общих принципов стандартных клиентских веб-приложений. Понимая внутренний механизм действия JavaScript и разбираясь в базовых интерфейсах API, предоставляемых браузерами (например, для обработки событий, у которых много общего с механизмами, предоставляемыми в среде Node.js), Анна может значительно укрепить и расширить свои навыки, как, впрочем, и вы. В ходе этого процесса вы можете стать более универсальным разработчиком, приобретя знания, необходимые для решения самых разных задач. Вы сможете даже написать собственные безсерверные приложения, полностью размещаемые в облаке, используя интерфейсы JavaScript API (например, AWS Lambda) для развертывания, сопровождения и управления облачными компонентами своего приложения.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Клиентские веб-приложения относятся к числу самых распространенных, а понятия, инструментальные средства и методики, применявшиеся некогда для их разработки, проникли в другие прикладные области. Ясное понимание основ разработки клиентских веб-приложений поможет вам разрабатывать приложения для самых разных предметных областей.
- Чтобы усовершенствовать навыки разработки, вам придется основательно разобраться во внутреннем механизме действия JavaScript, а также в инфраструктуре, предоставляемой браузерами.

- В данной книге основное внимание уделяется таким механизмам JavaScript, как функции, замыкания функций и прототипы, а также новым языковым средствам JavaScript, в том числе генераторам, обещаниям, отображениям, множествам и модулям.
- Код JavaScript может выполняться в самых разных средах, но прежде всего в браузере – среде, с которой все начинается и которой будет уделено основное внимание в данной книге.
- Помимо JavaScript, в данной книге будут исследованы внутренние механизмы работы браузеров, в том числе модель DOM (структурированное представление пользовательского интерфейса веб-страницы) и события, поскольку клиентские веб-приложения управляются событиями.
- Подобные исследования будут проведены в данной книге с учетом норм передовой практики программирования: отладки, тестирования и анализа производительности прикладного кода.

2 Создание страницы в динамическом режиме

В этой главе...

- Стадии жизненного цикла веб-приложения
- Обработка кода HTML для получения веб-страницы
- Порядок выполнения кода JavaScript
- Достижение согласованности действий с помощью событий
- Цикл ожидания событий

Наше исследование JavaScript выполняется в контексте клиентских веб-приложений и браузера в качестве интерпретатора, выполняющего код JavaScript. Чтобы заложить прочное основание, опираясь на которое можно продолжить исследование JavaScript как языка и браузера как платформы, мы должны разъяснить сначала весь жизненный цикл веб-приложения и выяснить, каким образом код JavaScript вписывается в этот жизненный цикл.

В этой главе жизненный цикл клиентских веб-приложений будет тщательно исследован, начиная с момента запроса страницы, продолжая различными взаимодействиями с пользователем и кончая закрытием веб-страницы. Сначала мы выясним, каким образом страница создается в результате обработки кода HTML. Затем уделим внимание выполнению кода JavaScript, который в основном придает динамичность страницам. И, наконец, рассмотрим порядок обработки событий, чтобы стало понятно, как разрабатывать интерактивные приложения, реагирующие на действия пользователей.

В ходе этого процесса мы исследуем такие понятия, составляющие основу веб-приложений, как модель DOM (структурированное представление

веб-страницы) и цикл ожидания событий, определяющий порядок их обработки в приложениях.

Знаете ли вы?

Всегда ли браузер формирует страницу точно по заданной HTML-разметке?

Сколько событий может одновременно обрабатывать веб-приложение?

Почему в браузерах должна быть организована очередь событий для их обработки?

2.1. Общее представление о жизненном цикле веб-приложения

Жизненный цикл типичного клиентского веб-приложения начинается с ввода пользователем URL в поле адреса или щелчка на ссылке в окне браузера. Допустим, требуется найти какой-нибудь термин, перейдя на начальную страницу поисковой машины Google. С этой целью в поле адреса вводится URL (`www.google.com`) ❶, как показано на рис. 2.1, *слева вверху*.

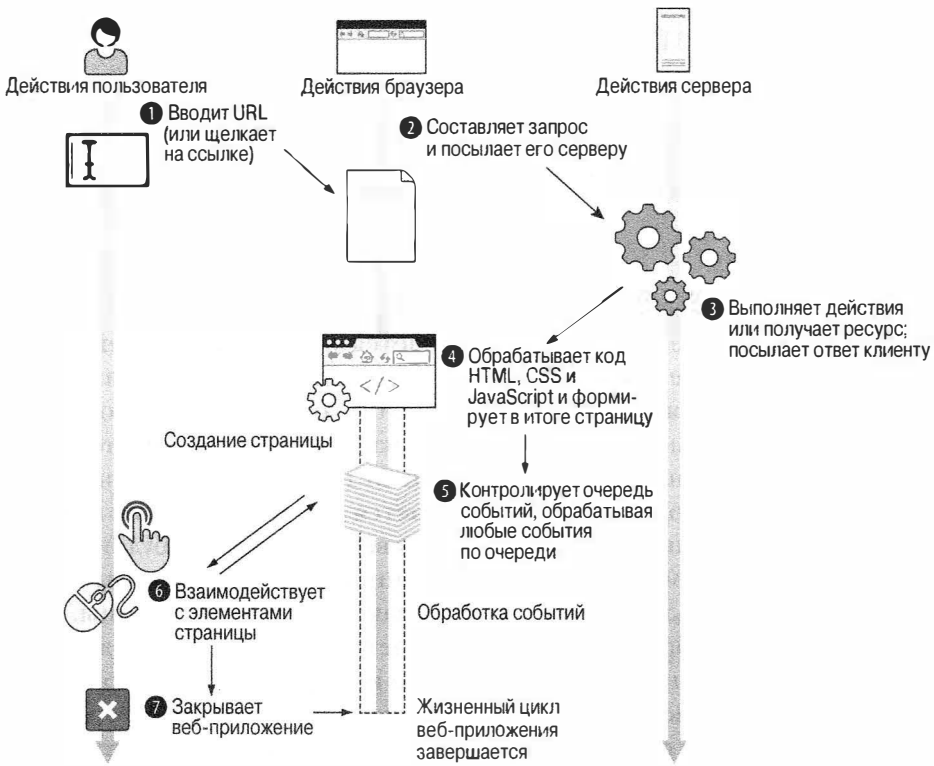


Рис. 2.1. Жизненный цикл клиентского веб-приложения начинается с ввода пользователем адреса веб-сайта (или щелчка на ссылке) и завершается, когда пользователь покидает веб-страницу. Он состоит из двух стадий: *создания страницы* и *обработки событий*

От имени пользователя браузер составляет запрос, посылаемый серверу ❷, который обрабатывает запрос ❸ и составляет ответ — как правило, из HTML-разметки, CSS-правил стилевого оформления и кода JavaScript. Именно в тот момент, когда браузер получает ответ ❹, и начинается жизненный цикл клиентского веб-приложения.

Клиентские веб-приложения относятся к типу приложений с графическим пользовательским интерфейсом (ГПИ), поэтому их жизненный цикл повторяет те же самые стадии, что и у приложений с ГПИ (настольных или для мобильных устройств). В частности, он выполняется в следующие две стадии.

- **Создание страницы.** Установка пользовательского интерфейса.
- **Обработка событий.** Вход в цикл ожидания событий ❺, при наступлении которых вызываются обработчики событий ❻.

Жизненный цикл веб-приложения завершается в тот момент, когда пользователь закрывает или покидает веб-страницу ❼.

А теперь рассмотрим пример веб-приложения с простым ГПИ, реагирующего на действия пользователя. Всякий раз, когда пользователь перемещает мышь или щелкает кнопкой мыши на странице, отображается сообщение. Данным примером веб-приложения мы будем пользоваться на протяжении всей этой главы.

Листинг 2.1. Небольшое веб-приложение с ГПИ, реагирующее на события

```
<!DOCTYPE html>
<html>
  <head>
    <title>Web app lifecycle</title>
    <style>
       #first { color: green;}
       #second { color: red;}
    </style>
  </head>
  <body>
    <ul id="first"></ul>

    <script>
      function addMessage(element, message){
        var messageElement = document.createElement("li");
        messageElement.textContent = message;
        element.appendChild(messageElement);
      }

      var first = document.getElementById("first");
      addMessage(first, "Page loading");
    </script>

    <ul id="second"></ul>

    <script>
        document.body.addEventListener("mousemove", function() {
```

Определить функцию, записывающую сообщение в элемент

Подключить обработчик событий перемещения мыши к телу документа

```
        var second = document.getElementById("second");
        addMessage(second, "Event: mousemove");
      });

      document.body.addEventListener("click", function(){
        var second = document.getElementById("second");
        addMessage(second, "Event: click");
      });
    </script>
  </body>
</html>
```

Подключить обработчик событий от щелчков кнопкой мыши к телу документа

Сначала в листинге 2.1 определяются два CSS-правила стилевого оформления (`#first` и `#second`), которые обозначают цвет текста в элементах разметки идентификаторами `first` и `second`, чтобы их можно было легко различать. Затем элемент разметки списка с идентификатором `first` определяется следующим образом:

```
<ul id="first"></ul>
```

Далее определяется функция `addMessage()`, при вызове которой создается новый элемент маркированного списка, задается его текстовое содержимое, а затем он добавляется к существующему элементу разметки:

```
function addMessage(element, message){
  var messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
}
```

Вслед за этим вызывается встроенный метод `getElementById()` для извлечения элемента с идентификатором `first` из документа, а затем в него записывается сообщение, извещающее о загрузке страницы, как показано ниже.

```
var first = document.getElementById("first");
addMessage(first, "Page loading");
```

Далее определяется еще один элемент разметки списка, но на этот раз — с идентификатором `second`:

```
<ul id="second"></ul>
```

И, наконец, к телу веб-страницы подключаются два обработчика событий. Сначала добавляется обработчик событий `mousemove`, который выполняется всякий раз, когда пользователь перемещает мышь. В этом обработчике вызывается функция `addMessage()` для записи сообщения `"Event: mousemove"` в элемент разметки списка `second`:

```
document.body.addEventListener("mousemove", function() {
  var second = document.getElementById("second");
  addMessage(second, "Event: mousemove");
});
```

Затем регистрируется обработчик событий `click`, который выполняется всякий раз, когда пользователь щелкает кнопкой мыши на странице. В нем также выводятся сообщения `"Event: click"` в элемент разметки списка `second`, как показано ниже.

```
document.body.addEventListener("click", function(){
  var second = document.getElementById("second");
  addMessage(second, "Event: click");
});
```

На рис. 2.2 приведен результат выполнения данного веб-приложения при его взаимодействии с пользователем.

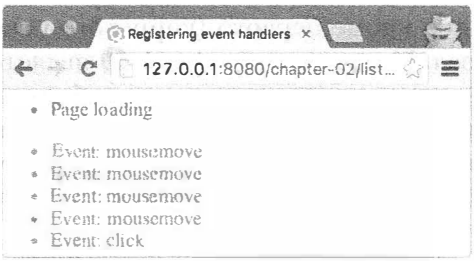


Рис. 2.2. При выполнении кода из листинга 2.1 на странице выводятся разные сообщения в зависимости от действий пользователя

В следующих разделах мы воспользуемся этим примером для исследования и демонстрации различия между стадиями жизненного цикла веб-приложения. А теперь давайте перейдем к этапу создания веб-страницы.

2.2. Стадия создания страницы

Чтобы стало возможным взаимодействие с пользователем или отображение данного веб-приложения, необходимо сначала создать страницу, исходя из информации, получаемой от сервера в ответе, как правило, состоящем из HTML-разметки, CSS-правил стилевого оформления и кода JavaScript. Назначение этой стадии — установить пользовательский интерфейс веб-приложения, и это делается в два отдельных этапа.

1. Синтаксический анализ HTML-разметки и построение объектной модели документа (DOM).
2. Выполнение кода JavaScript.

На первом этапе браузер обрабатывает узлы HTML-разметки документа, а на втором этапе выполняется сценарий JavaScript всякий раз, когда в HTML-разметке встречается специальный элемент `script`, содержащий код JavaScript или ссылку на него. На стадии создания страницы браузер может переходить от одного из этих этапов к другому столько раз, сколько потребуется (рис. 2.3).

2.2.1. Синтаксический анализ кода HTML и построение модели DOM

Стадия создания страницы начинается с получения браузером кода HTML, на основе которого он строит пользовательский интерфейс страницы. С этой целью браузер производит синтаксический анализ кода HTML по отдельным элементам HTML-разметки и строит модель DOM — структурированное представление HTML-страницы, где каждый элемент HTML-разметки представлен отдельным узлом дерева. В качестве примера на рис. 2.4 показана модель DOM страницы, построенная вплоть до появления элемента разметки `script`.

Обратите внимание на то, что узлы на рис. 2.4 организованы таким образом, чтобы у каждого из них, кроме первого (корневого узла `html` ❶) был лишь один родительский узел. Например, узел `html` ❶ является родительским для узла `head` ❷. В то же время у каждого узла может быть любое количество дочерних узлов, например, узлы `head` ❷ и `body` ❼, являются дочерними для узла `html` ❶. Узлы, порожденные от одного и того же элемента, называются *родственными*. Так, узлы `head` ❷ и `body` ❼ являются родственными.

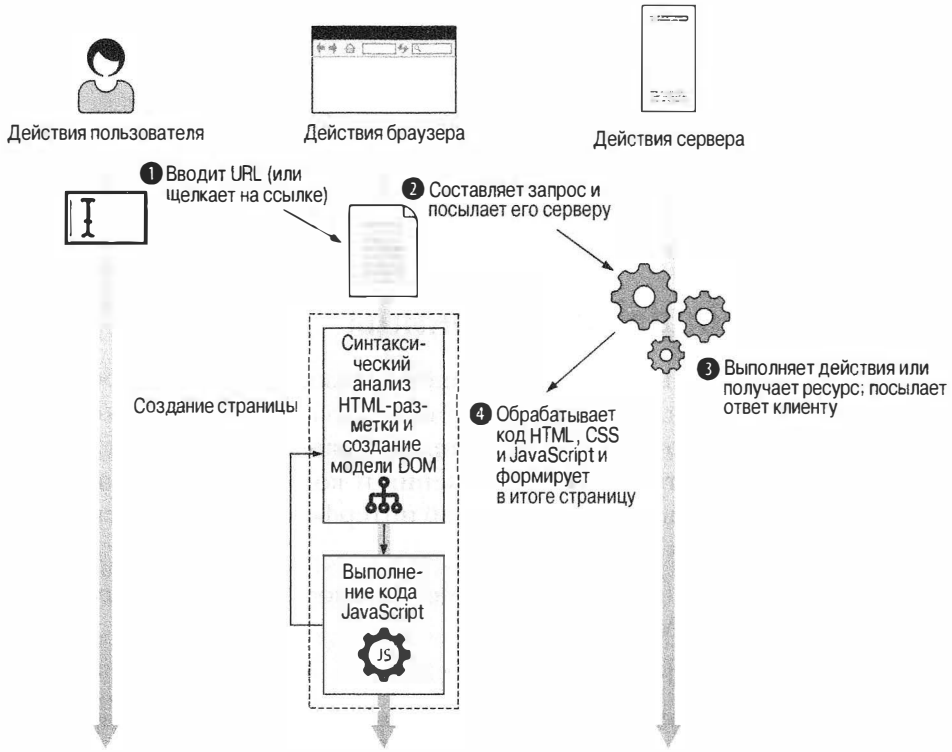


Рис. 2.3. Стадия создания страницы начинается в тот момент, когда браузер получает код страницы. Она происходит в два этапа: сначала синтаксический анализ HTML-разметки, а затем построение модели DOM и выполнение кода JavaScript

Следует особо подчеркнуть, что HTML-разметка и модель DOM — не одно и то же, хотя они и тесно связаны, причем последняя строится на основании первой. Код HTML следует рассматривать в качестве *образца*, по которому браузер строит первоначальную модель DOM, а по существу, —пользовательский интерфейс страницы. Браузер может даже исправить ошибки, обнаруживаемые в таком образце, чтобы построить правильную модель DOM. Рассмотрим наглядный тому пример, приведенный на рис. 2.5.

На рис. 2.5 приведен простой пример некорректного кода HTML, где элемент разметки абзаца размещается в элементе разметки `head`. Ведь элемент разметки `head` предназначен для предоставления общей информации о странице, например, ее заголовка, кодировки символов, внешних файлов со стилевыми таблицами и сценариев, но не для определения содержимого страницы, как в данном примере. Такая HTML-разметка считается ошибочной, и поэтому браузер "молча" ее исправляет, строя правильную модель DOM, как показано рис. 2.5, *справа*. В этой модели элемент разметки абзаца размещается в элементе разметки `body`, где и должно находиться содержимое страницы.

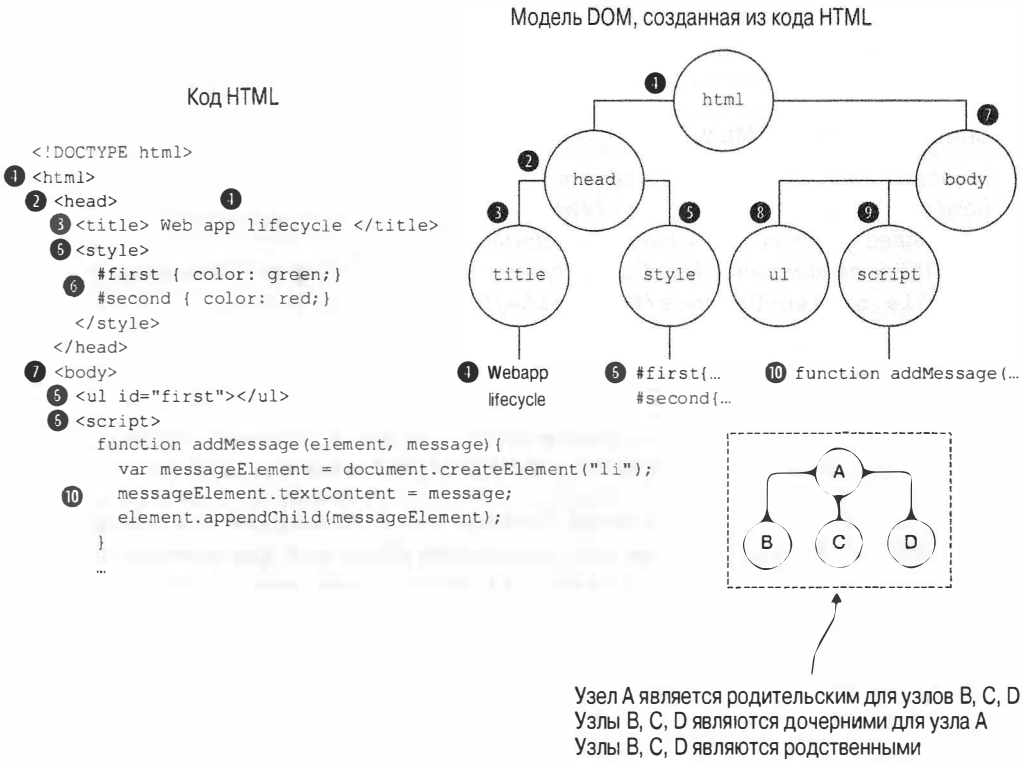


Рис. 2.4. К тому моменту, когда браузер обнаружит первый элемент разметки **`script`**, он уже построит модель DOM из нескольких элементов HTML-разметки (узлы *справа*)

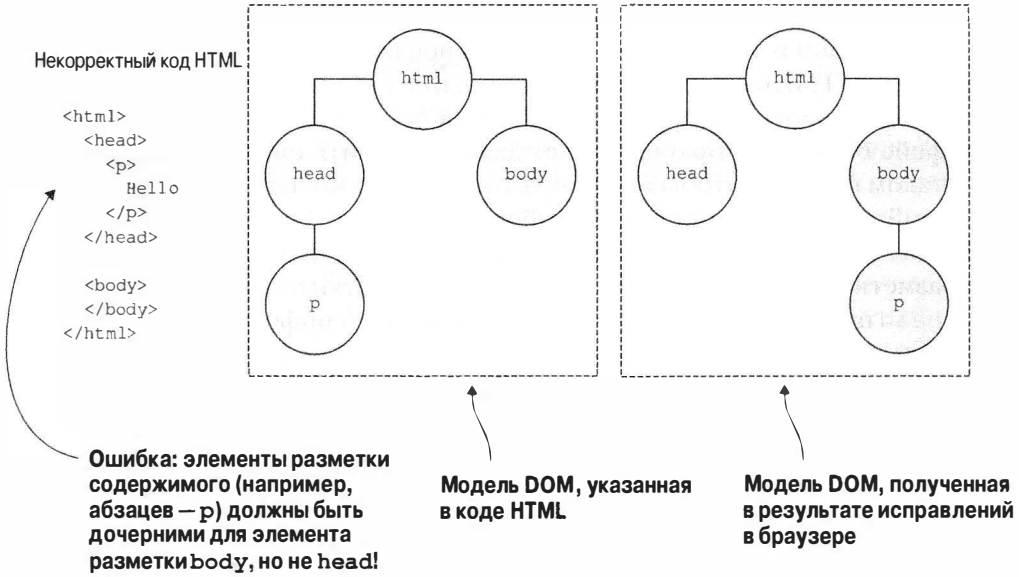


Рис. 2.5. Пример некорректной HTML-разметки, исправленной браузером

Спецификации HTML и DOM

В настоящее время действует версия HTML5 языка HTML, спецификация которого доступна по адресу `https://html.spec.whatwg.org/`. Если же требуется более удобная для чтения документация, то рекомендуется руководство по HTML5 от компании Mozilla, доступное по адресу `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5`.

С другой стороны, развитие модели DOM происходит медленнее. В настоящее время действует версия DOM3, спецификация которой доступна по адресу `https://dom.spec.whatwg.org/`. И в этом случае компания Mozilla подготовила более удобную для чтения документацию, доступную по адресу `https://developer.mozilla.org/ru/docs/DOM/DOM_Reference`.

В процессе создания страницы браузер может обнаружить специальный тип элемента HTML-разметки `script`, который служит для включения кода JavaScript в состав страницы. Когда в HTML-разметке встречается такой элемент, браузер приостанавливает построение модели DOM из кода HTML и начинает выполнение кода JavaScript.

2.2.2. Выполнение кода JavaScript

Весь код JavaScript, содержащийся в элементе разметки `script`, выполняется встроенным в браузер интерпретатором JavaScript, например, движком Spidermonkey в браузере Firefox, V8 — в браузерах Chrome и Opera или Chakra — в браузере Microsoft Edge (или Internet Explorer). Основное назначение кода

JavaScript — обеспечить динамичность страницы, и с этой целью браузер предоставляет интерфейс API через глобальный объект, который может быть использован интерпретатором JavaScript для взаимодействия со страницей и ее видоизменения.

Глобальные объекты в JavaScript

Исходно браузер делает доступным для интерпретатора JavaScript глобальный объект `window`, представляющий окно, в котором отображается страница. Объект `window` является *исходным* глобальным объектом, через который становятся доступными все остальные глобальные объекты, переменные (даже те, что определяются пользователем) и интерфейсы API браузера. К числу самых важных свойств глобального объекта `window` относится объект `document`, представляющий модель DOM текущей страницы. Используя этот объект в коде JavaScript, можно изменить модель DOM до любой степени, видоизменяя или удаляя существующие в ней элементы и даже создавая и вводя новые.

Рассмотрим следующую строку кода из листинга 2.1:

```
var first = document.getElementById("first");
```

В этой строке кода глобальный объект `document` используется для выбора элемента с идентификатором `first` из модели DOM и его присваивания переменной `first`. Благодаря этому в коде JavaScript можно далее произвести всевозможные модификации данного элемента, в том числе изменить его текстовое содержимое, видоизменить его атрибуты, динамически создать и ввести в него новые порожденные элементы и даже удалить элемент из модели DOM.

Интерфейсы API браузеров

На протяжении всей этой книги мы будем пользоваться рядом встроенных в браузер объектов и функций (например, глобальными объектами **`window`** и **`document`**). К сожалению, в этой книге нельзя охватить все, что поддерживается браузерами, поскольку она посвящена языку JavaScript. Правда, компания Mozilla предоставляет удобную документацию по адресу **`https://developer.mozilla.org/ru/docs/Web/API`**, где можно ознакомиться с современным состоянием интерфейсов API браузера для разработки веб-приложений.

Итак, дав общее понятие о глобальных объектах браузера, перейдем к рассмотрению двух других типов кода JavaScript, которые определяют, когда именно следует выполнять этот код.

Разные типы кода JavaScript

В целом различают два типа кода JavaScript: *глобальный код* и *код функции*. Пример, приведенный в листинге 2.2, поможет лучше понять отличия этих двух типов кода.

Главное отличие этих двух типов кода заключается в месте их расположения. В частности, код, содержащийся в теле функции, называется *кодом функции*, а код, располагающийся за пределами всех функций, — *глобальным кодом*.

Листинг 2.2. Глобальный код и код функции

```
<script>
  function addMessage(element, message){
    var messageElement = document.createElement("li");
    messageElement.textContent = message;
    element.appendChild(messageElement);
  }

  var first = document.getElementById("first");
  addMessage(first, "Page loading");
</script>
```

Код функции находится в теле функции

Глобальный код находится за пределами функции

Эти два типа кода отличаются также порядком их выполнения, а дополнительные их отличия будут рассмотрены далее, особенно в главе 5. Глобальный код выполняется интерпретатором JavaScript (подробнее об этом речь пойдет несколько позже) автоматически строка за строкой по мере его появления. Например, в листинге 2.2 приведены фрагменты глобального кода, где сначала определяется функция `addMessage()`, а затем вызывается встроенный метод `getElementById()` для извлечения элемента по идентификатору `first` и далее функция `addMessage()`. Эти фрагменты глобального кода выполняются один за другим по мере их появления, как показано на рис. 2.6.

С другой стороны, для выполнения кода функции придется сделать еще кое-что: выполнить глобальный код (например, вызвать функцию `addMessage()` в глобальном коде, чтобы выполнить содержащийся в ней код функции), вызвать какую-нибудь другую функцию или же сделать вызов из браузера, как поясняется далее.

Выполнение кода JavaScript на стадии создания страницы

Когда браузер достигает узла `script` на стадии создания страницы, он приостанавливает построение модели DOM на основании кода HTML и вместо этого приступает к выполнению кода JavaScript. Это означает, что выполняется глобальный код JavaScript, содержащийся в элементе разметки `script`, а также функции, вызываемые из глобального кода.

Рис. 2.6. Порядок выполнения программы при исполнении кода JavaScript

Вернемся к примеру из листинга 2.1. На рис. 2.7 приведено состояние модели DOM после выполнения глобального кода JavaScript. Проанализируем подробно его выполнение. Сначала в этом коде определяется приведенная ниже функция `addMessage()`.

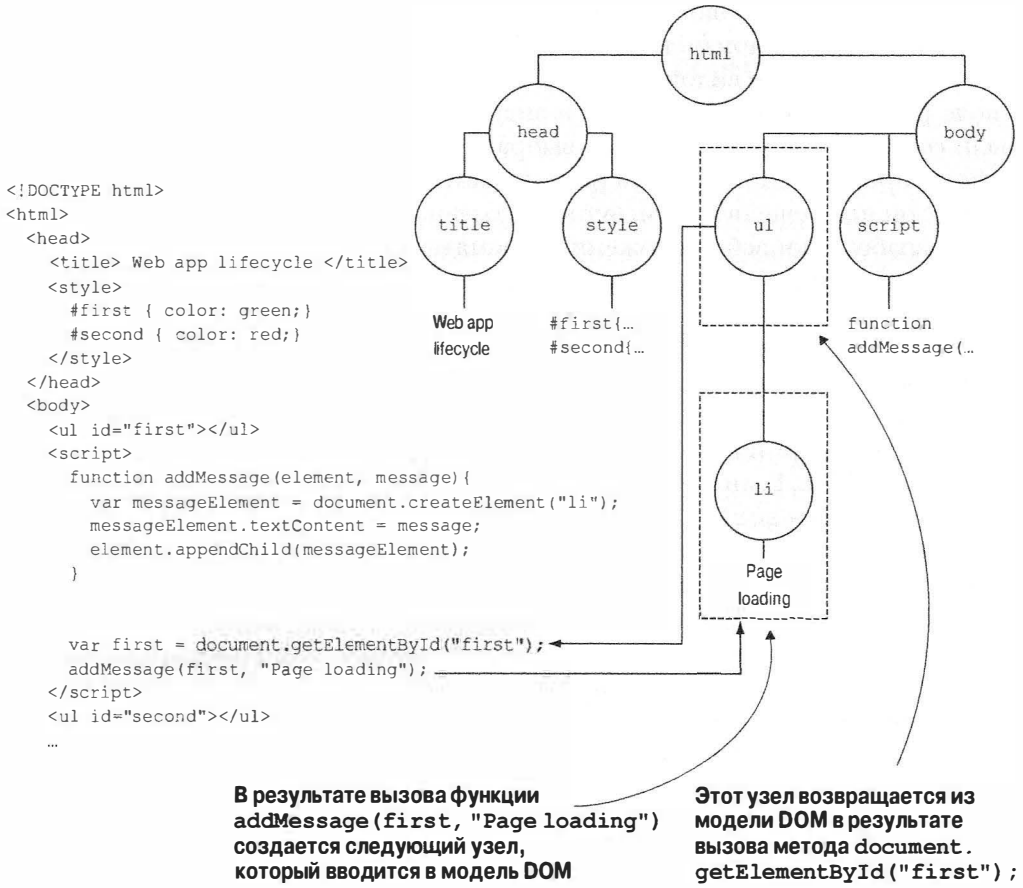


Рис. 2.7. Состояние модели DOM страницы после выполнения кода JavaScript, содержащегося в элементе разметки `script`

```
function addMessage(element, message){
  var messageElement = document.createElement("li");
  messageElement.textContent = message;
  element.appendChild(messageElement);
}
```

Затем существующий элемент извлекается из модели DOM с помощью глобального объекта `document` и его метода `getElementById()`:

```
var first = document.getElementById("first");
```

Далее вызывается функция `addMessage()`:

```
addMessage(first, "Page loading");
```

что приводит к созданию нового элемента разметки `li`, видоизменению его текстового содержимого, а в конечном итоге — к его вводу в модель DOM.

В данном примере кода JavaScript текущая модель DOM видоизменяется в результате создания нового элемента и его ввода в эту модель. А в общем, модель DOM можно видоизменить в коде JavaScript до любой степени, в том числе создать новые узлы и видоизменить или удалить существующие узлы DOM. Но в коде JavaScript нельзя выбирать и видоизменять элементы, которые еще не были созданы. Например, нельзя выбрать или видоизменить элемент с идентификатором `second`, который располагается после текущего узла `script`, поскольку он пока еще не достигнут и не создан. И это одна из причин, по которым разработчики веб-приложений стремятся размещать элементы разметки в самом низу страницы. Благодаря этому исключается проблема доступа к отдельным элементам HTML-разметки страницы.

Как только интерпретатор JavaScript выполнит последнюю строку кода JavaScript из элемента разметки `script` (а это означает возврат из функции `addMessage()`, как показано на рис. 2.5), браузер выйдет из режима выполнения кода JavaScript и продолжит построение узлов DOM, обрабатывая оставшийся код HTML. Если же в процессе этой обработки браузер снова обнаружит элемент разметки `script`, построение модели DOM опять приостановится и интерпретатор JavaScript приступит к выполнению кода JavaScript, содержащегося в этом элементе. Следует особо подчеркнуть, что глобальное состояние JavaScript-приложения между тем сохраняется. Все глобальные переменные, определенные пользователем во время выполнения кода JavaScript из одного элемента разметки `script`, как правило, доступны для кода JavaScript из других элементов разметки `script`. И это происходит благодаря глобальному объекту `window`, который хранит все глобальные переменные JavaScript, действует и доступен в течение всего жизненного цикла страницы.

Таким образом, оба этапа построения модели DOM из HTML-разметки и кода JavaScript повторяются до обработки всех элементов HTML-разметки и выполнения всех строк кода JavaScript.

И, наконец, когда браузер обработает все элементы HTML-разметки, стадия создания страницы завершается. После этого браузер переходит ко второй стадии жизненного цикла веб-приложения: *обработке событий*.

2.3. Обработка событий

Клиентские веб-приложения относятся к типу приложений с ГПИ, а это означает, что они реагируют на самые разные события, наступающие при перемещении мыши, щелчках кнопками мыши, нажатии клавиш и т.д. Именно поэтому в коде JavaScript, выполняемом на стадии создания страницы, можно не только оказывать влияние на глобальное состояние приложения и видоизменять модель DOM, но и регистрировать обработчики (или слушатели) со-

бытий – функции, которые выполняются браузером при наступлении соответствующих событий. С помощью этих обработчиков событий обеспечивается интерактивность веб-приложений. Но прежде чем перейти непосредственно к вопросу регистрации обработчиков событий, представим общие принципы обработки событий.

2.3.1. Общее представление об обработке событий

Среда выполнения браузера, по существу, действует по принципу поочередного выполнения фрагментов кода в соответствии с моделью так называемого *однопоточного* выполнения. В качестве аналогии рассмотрим очередь клиентов в банке. Все клиенты становятся в один ряд и должны ждать своей очереди на обслуживание банковскими служащими. Но ведь в банке JavaScript открыто только одно кассовое окно! И через него обслуживается только один клиент, который почему-то считает, что все планирование своих финансов на очередной бюджетный год он может сделать прямо у кассового окна, не считаясь со временем других клиентов, стоящих в очереди, и тем самым стопоря всю работу банка.

Всякий раз, когда наступает событие, браузер должен выполнить функцию соответствующего обработчика событий. Но при этом не гарантируется, что пользователи будут терпеливо ждать наступления очередного события. Именно поэтому браузеру нужно каким-то образом отслеживать наступившие, но еще не обработанные события. И для этой цели в браузере организуется *очередь событий*, как показано на рис. 2.8.

Все инициируемые события, будь то пользователем (при перемещении мыши или нажатии клавиш) или сервером (Ajax-события), ставятся в одну и ту же очередь событий в том порядке, в каком они обнаруживаются браузером. Как показано на рис. 2.8, *посредине*, процесс обработки событий может быть описан по следующей простой схеме.

- Браузер отслеживает содержимое начала очереди событий.
- Если события отсутствуют, браузер ожидает их появления.
- Если в начале очереди присутствует событие, браузер извлекает его из очереди и запускает соответствующий обработчик, если таковой имеется. Во время выполнения кода этого обработчика остальные события терпеливо ожидают своей очереди на обработку.

В связи с тем что одновременно может быть обработано только одно событие, приходится уделять особое внимание количеству времени, отводимому для обработки событий. Ведь если написать обработчики событий, выполнение которых отнимает немало времени, то веб-приложение не будет вовремя реагировать на внешние воздействия! (Если это объяснение покажется вам не совсем понятным, не отчаивайтесь – мы еще вернемся к циклу ожидания событий в главе 13 и разъясним подробно, как он оказывает влияние на воспринимаемую производительность веб-приложений.)

Следует особо подчеркнуть, что механизм браузера, размещающий события в очереди, действует внешним образом по отношению к стадиям создания страниц и обработки событий. Операция, которая требуется для определения момента наступления событий и постановки их в очередь, не выполняется в потоке, где обрабатываются события.

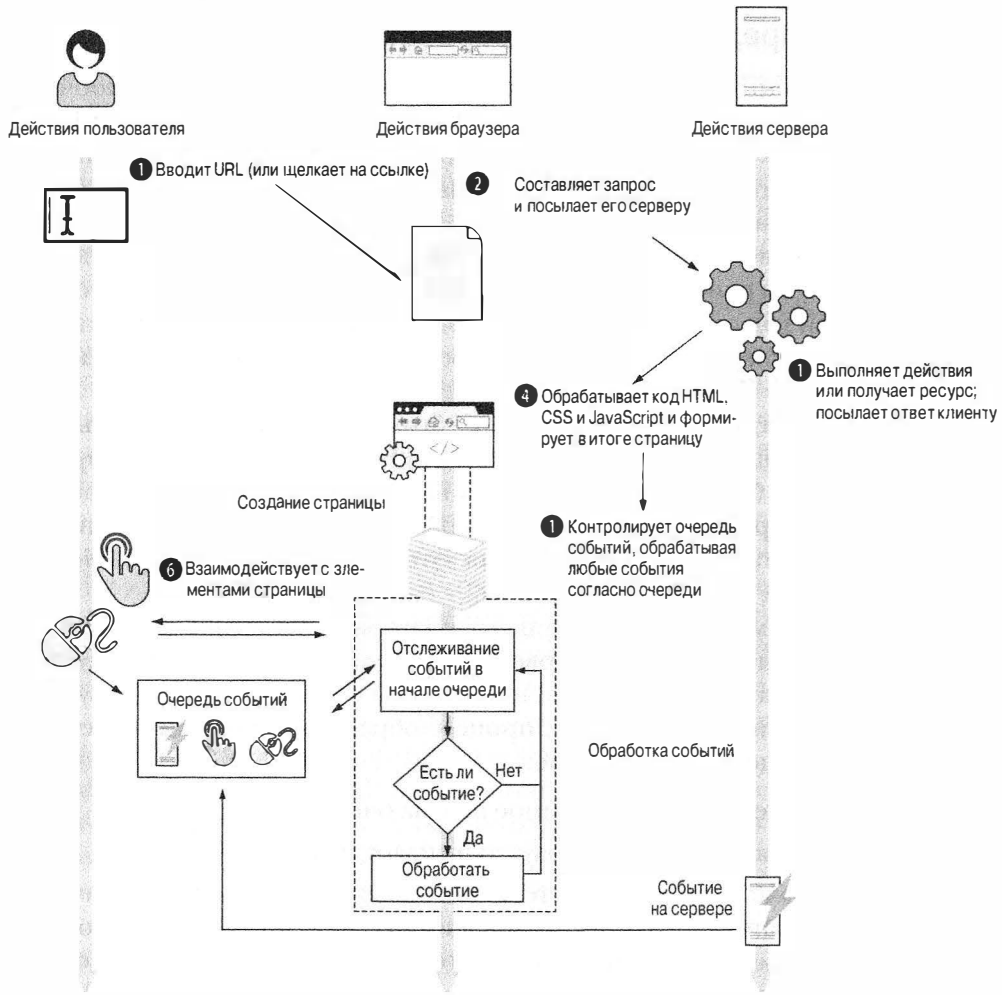


Рис. 2.8. На стадии обработки событий все события независимо от их источника, будь то пользователь, щелкающий кнопкой мыши или нажимающий клавишу на клавиатуре, или сервер, инициирующий Ajax-события, ставятся в очередь по мере их наступления и обрабатываются по мере возможности в единственном потоке исполнения

Асинхронный характер событий

События могут наступать в непредсказуемые моменты времени и в непредсказуемом порядке. (Попробуйте заставить пользователей нажать клавиши или

щелкнуть кнопками мыши в каком-то определенном порядке!) Именно поэтому и говорят, что обработка событий, а следовательно, и вызов функций их обработки происходит *асинхронно*.

Среди прочего могут произойти следующие типы событий.

- События в браузере, в том числе по окончании загрузки страницы или перед ее выгрузкой.
- События в сети, например, при ответах, поступающих от сервера (Ajax-события, серверные события).
- События, инициируемые пользователем, когда он, например, щелкает кнопкой мыши, перемещает мышь или нажимает клавиши на клавиатуре.
- События, инициируемые таймером, в том числе по истечении времени ожидания или заданного промежутка времени.

Большая часть кода веб-приложений запускается в результате наступления подобных событий!

Принцип обработки событий занимает центральное место в разработке веб-приложений, и поэтому он не раз демонстрируется в примерах, приведенных в данной книге. Код обработки событий назначается заранее и запускается в дальнейшем при наступлении события. Большая часть кода, размещаемого на странице, кроме глобального, будет запускаться и выполняться в результате наступления какого-нибудь события.

Прежде чем обрабатывать события, код приложения должен уведомить браузер, что его интересует обработка конкретных событий. Поэтому рассмотрим, каким образом регистрируются обработчики событий.

2.3.2. Регистрация обработчиков событий

Как упоминалось выше, обработчики событий являются функциями, которые требуется выполнять всякий раз, когда наступает конкретное событие. Чтобы это произошло, нужно каким-то образом уведомить браузер, что нас интересует данное событие. И такой процесс называется *регистрацией обработчиков событий*. Обработчики событий регистрируются в клиентских веб-приложениях следующими способами.

- Присваиванием функций специальным свойствам.
- Вызовом встроенного метода `addEventListener()`.

Например, в следующей строке кода функция присваивается специальному свойству `onload` объекта `window`:

```
window.onload = function(){};
```

В данном случае регистрируется обработчик событий типа `load`, наступающих в тот момент, когда модель DOM полностью построена и готова к применению. (Не обращайте пока что особого внимания на необычный вид правой

части операции присваивания. Более подробно функции будут рассматриваться в последующих главах.)

```
document.body.onclick = function(){};
```

Присваивание функций специальным свойствам служит простым и удобным способом регистрации обработчиков событий, и вам, вероятно, уже приходилось наблюдать его применение в прикладном коде. Тем не менее мы *не* рекомендуем регистрировать обработчики событий подобным способом, поскольку ему присущ следующий недостаток: зарегистрировать можно только одну функцию для обработки конкретного события. Это означает, что можно неумышленно перезаписать предыдущие функции обработки событий, даже не заметив этого. Правда, имеется другой способ: вызвать метод `addEventListener()`, позволяющий зарегистрировать столько функций обработки событий, сколько потребуется. В качестве примера такого способа регистрации обработчиков событий в листинге 2.3 приведен фрагмент кода из листинга 2.1.

Листинг 2.3. Регистрация обработчиков событий

```
<script>
  document.body.addEventListener("mousemove", function() {
    var second = document.getElementById("second");
    addMessage(second, "Event: mousemove");
  });

  document.body.addEventListener("click", function(){
    var second = document.getElementById("second");
    addmessage(second, "Event: click");
  });
</script>
```

Зарегистрировать обработчик событий перемещения мыши

Зарегистрировать обработчик событий от щелчков кнопками мыши

В данном примере встроенный метод `addEventListener()` вызывается для элемента HTML-разметки с указанием типа события (`mousemove` или `click`) и регистрируемой функции обработки событий. Это означает, что всякий раз, когда курсор мыши перемещается по странице, браузер вызывает функцию, выводящую текстовое сообщение `"Event: mousemove"` в элемент списка с идентификатором `second`. Аналогичным образом сообщение `"Event: click"` вводится в тот же самый список всякий раз, когда на странице выполняется щелчок кнопкой мыши.

А теперь, когда вы ознакомились со способами регистрации обработчиков событий, обратимся снова к упоминавшейся выше схеме, чтобы подробнее рассмотреть порядок обработки событий. Главный принцип обработки событий состоит в следующем: когда наступает событие, браузер вызывает соответствующий обработчик. Но, как упоминалось выше, одновременно может быть выполнен только один обработчик событий вследствие того, что в среде браузера применяется модель однопоточного выполнения. Любые последующие события обрабатываются только после того, как завершится выполнение текущего обработчика событий!

Вернемся к примеру веб-приложения из листинга 2.1. На рис. 2.9 приведен пример выполнения данного приложения на стадии, когда проворный пользователь переместил мышь и щелкнул кнопкой мыши.

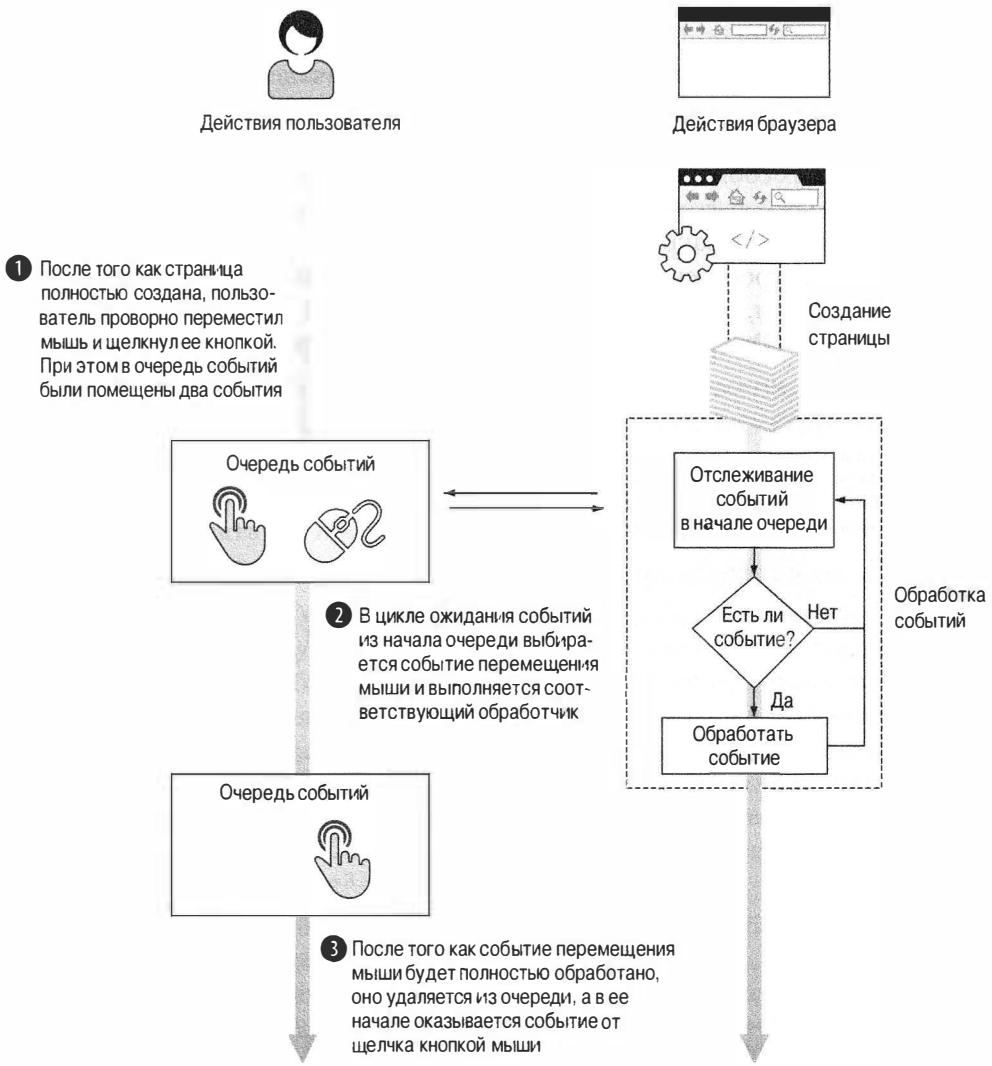


Рис. 2.9. Пример стадии обработки событий **mousemove** и **click**

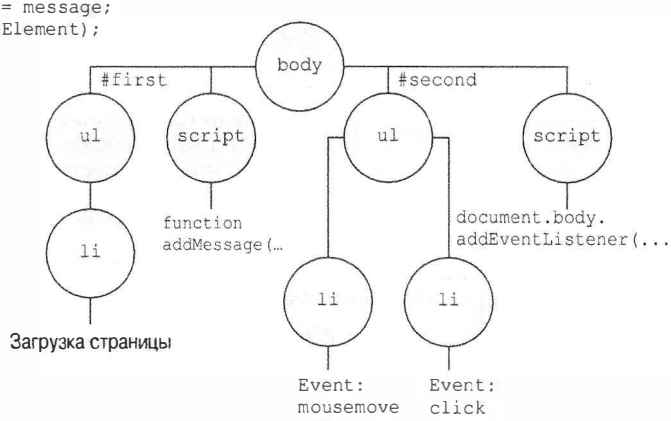
Рассмотрим подробнее, что же в данном случае происходит. В качестве реакции на подобные действия пользователя браузер помещает события `mousemove` и `click` в очередь событий в том порядке, в каком они наступают: сначала событие `mousemove`, а затем событие `click` ❶.

На стадии обработки событий в цикле ожидания событий проверяется их очередь и обнаруживается, что в ее начале находится событие `mousemove`, в

результате чего выполняется соответствующий обработчик событий ❷. В то время, как выполняется обработчик событий типа `mousemove`, событие `click` ожидает своей очереди на обработку. Как только будет выполнена последняя строка кода в функции обработки событий типа `mousemove`, событие данного типа окажется полностью обработанным ❸, и в цикле ожидания событий будет снова проверена их очередь. На этот раз в начале очереди обнаруживается событие `click`, которое и обрабатывается. По окончании выполнения обработчика событий типа `click` в очереди больше не остается ожидающих обработки новых событий, и рассматриваемый здесь цикл выполняется далее в ожидании новых событий. Выполнение этого цикла продолжается до тех пор, пока пользователь не закроет веб-приложение.

Итак, рассмотрев основные этапы, происходящие на стадии обработки событий, выясним, каким образом выполнение этой стадии оказывает влияние на модель DOM (рис. 2.10). При выполнении обработчика событий типа

```
<ul id="first"></ul>

<script>
  function addMessage(element, message){
    var messageElement = document.createElement('li');
    messageElement.textContent = message;
    element.appendChild(messageElement);
  }
</script>
```




```
<ul id="second"></ul>

<script>
  document.body.addEventListener('mousemove',function(){
    var second = document.getElementById('second');
    addMessage(second, 'Event: mousemove');
  });

  document.body.addEventListener('click',function(){
    var second = document.getElementById('second');
    addMessage(second, 'Event: click');
  });

</script>
```

Рис. 2.10. Состояние модели DOM из рассматриваемого здесь примера веб-приложения после обработки событий типа **`mousemove`** и **`click`**

mousemove выбирается второй элемент списка с идентификатором second и с помощью функции addMessage() в него вводится новый элемент списка ❶ с текстом сообщения "Event: mousemove". Как только выполнение обработчика событий типа mousemove завершится, в цикле ожидания событий начнется выполнение обработчика событий типа click. Это приведет к созданию еще одного элемента списка ❷, который также добавляется ко второму элементу списка с идентификатором second.

Вооружившись ясным пониманием жизненного цикла клиентского веб-приложения, в следующей части книги мы уделим основное внимание языку JavaScript, подробно исследовав все особенности функций.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Код HTML, получаемый браузером, служит в качестве образца для построения модели DOM – внутреннего представления структуры клиентского веб-приложения.
- Код JavaScript служит для динамического видоизменения модели DOM, чтобы придать веб-приложениям динамический характер их поведения.
- Выполнение клиентского веб-приложения происходит в две стадии.
  - **Создание страницы.** На этой стадии обрабатывается код HTML для построения модели DOM, а когда встречаются узлы сценариев, выполняется глобальный код JavaScript. В процессе выполнения кода JavaScript текущая модель DOM может быть видоизменена до любой степени и даже могут быть зарегистрированы обработчики событий – функции, выполняемые при наступлении конкретных событий (например, по щелчку кнопкой мыши или нажатию клавиши). Чтобы зарегистрировать обработчик событий, достаточно вызвать встроенный метод addEventListener().
  - **Обработка событий.** Различные события обрабатываются по очереди в том порядке, в каком они инициированы. Стадия обработки событий основывается, главным образом, на очереди событий, где все события хранятся в том порядке, в каком они наступают. В цикле ожидания событий всегда проверяется начало очереди, и если в ней обнаруживается событие, то вызывается соответствующая функция обработки событий.

Упражнения

1. Каковы две стадии жизненного цикла клиентского веб-приложения?
2. Какое главное преимущество дает применение метода addEvent Listener() для регистрации обработчиков событий по сравнению с

присваиванием обработчика событий отдельному свойству элемента разметки?

3. Сколько событий можно обработать одновременно?
4. В каком порядке обрабатываются события, извлекаемые из очереди событий?

Часть II

Представление о функциях

Итак, вы мысленно настроились на обучение и ясно представляете среду, в которой выполняется код JavaScript. Теперь вы готовы к изучению самых основ того арсенала средств, который доступен для вас в JavaScript.

Из главы 3 вы узнаете все о самом важном и основополагающем понятии JavaScript, которым является не объект, а *функция*. В этой главе поясняются причины, по которым ясное представление о функциях JavaScript является ключом к разгадке секретов этого языка программирования.

В главе 4 будет продолжено углубленное исследование функций. Из нее вы узнаете, каким образом вызываются функции, а также усвоите все особенности неявных параметров, доступных при выполнении кода функции.

В главе 5 изучение функций переходит на следующий уровень сложности. В ней рассматриваются замыкания — одно из самых превратно понимаемых (а порой и неизвестных) понятий языка JavaScript. Из этой главы вы узнаете, что замыкания тесно связаны с областями видимости. Помимо замыканий, в этой главе особое внимание уделяется понятию области видимости в JavaScript.

Наше исследование функций завершится в главе 6, где мы обсудим совершенно новый тип функции-генератора. У такой функции имеется ряд специальных свойств, и она оказывается полезной при работе с асинхронным кодом.

Функции высшего порядка для начинающих: определения и аргументы

3

В этой главе...

- Краткий обзор особой роли функций
- Рассмотрение функций в качестве объектов высшего порядка
- Способы определения функции
- Особенности присваивания параметров функции

Перейдя к чтению этой части, посвященной основам JavaScript, вы, вероятно, будете несколько удивлены тем, что в ней сначала рассматриваются *функции*, а не объекты. Разумеется, мы уделим достаточно внимания и объектам в части III, но, по существу, мастер программирования на JavaScript отличается от середнячка, главным образом, ясным представлением о том, что JavaScript — это язык *функционального программирования*. От понимания этой особенности JavaScript зависит уровень сложности всего кода, который вам предстоит вообще написать на этом языке.

Если вы читаете эту книгу, то уже не относитесь к новичкам и, следовательно, в достаточной степени владеете основами обращения с объектами, а расширенные представления о них вы можете почерпнуть из главы 7. Но ясное представление о роли функций в JavaScript — это единственное и самое главное оружие, которым вы можете владеть. Именно поэтому данная и три последующие главы посвящены подробному исследованию роли функций в JavaScript.

Самое главное, что функции в JavaScript относятся к категории *объектов высшего порядка*. Они могут интерпретироваться в JavaScript как любые другие объекты и сосуществовать с ними. На них, как и на обычные типы данных в

JavaScript, можно ссылаться с помощью переменных, объявлять в виде литералов и даже передавать в качестве параметров другим функциям.

В этой главе мы рассмотрим сначала те отличия, которые вносит такая ориентация на функции, а также покажем, каким образом это поможет в написании более компактного и легко понимаемого кода, если определять функции непосредственно там, где они нужны. В этой главе исследуются также выгоды, которые можно извлечь из функций в качестве объектов высшего порядка при написании более эффективных функций. По ходу изложения материала будут представлены различные способы определения функций, включая и некоторые их новые типы, в том числе *стрелочные* функции, что поможет в написании более изящного кода. И, наконец, будут рассмотрены отличия параметров и аргументов функции, причем особое внимание мы уделим таким новым возможностям ES6, как оставшимся (rest) и стандартным (default) параметрам. Итак, начнем с рассмотрения некоторых преимуществ функционального программирования.

Знаете ли вы?

В каких случаях функции обратного вызова могут быть использованы синхронно или асинхронно?

Чем стрелочная функция отличается от функционального выражения?

Почему возникает потребность в применении стандартных (устанавливаемых по умолчанию) значений параметров функции?

3.1. Главное отличие JavaScript как языка функционального программирования

Одна из причин, по которым функции и понятие функционала имеют особое значение в JavaScript, заключается в том, что функция является основным модульным исполняемым блоком. За исключением встраиваемого сценария, выполняющегося на этапе построения веб-документа, весь код сценария, который требуется написать для веб-страниц, обычно размещается в функции.

Большая часть кода JavaScript выполняется в результате вызова функции, и поэтому наличие функций в качестве универсальных и эффективных конструкций дает немалые удобства и свободу действий при написании кода. В большей части данной книги поясняется, каким образом характер функций как объектов высшего порядка можно использовать с наибольшей выгодой.

Но прежде рассмотрим ряд действий, которые можно выполнять над объектами. В языке JavaScript объекты можно

- создавать с помощью литералов: `{}`;
- присваивать переменным, элементам массива и свойствам других объектов, как показано в следующем примере кода:

```
var ninja = {};                  // Присвоить новый объект переменной
ninjaArray.push({});             // Ввести новый объект в массив
ninja.data = {};                 // Присвоить новый объект свойству другого объекта
```

- передавать в качестве аргументов функциям, как показано в следующем примере кода:

```
function hide(ninja){
  ninja.visibility = false;
}
hide({});   // Вновь созданный объект передается функции в качестве аргумента
```

- возвращать в качестве значений из функций, как показано в следующем примере кода:

```
function returnNewNinja() {
  return {};   // Возвратить новый объект из функции
}
```

- наделять свойствами, которые можно динамически создавать и присваивать им значения, как показано в следующем примере кода:

```
var ninja = {};
ninja.name = "Hanzo";   // Создать новое свойство для объекта
```

Таким образом, в отличие от многих других языков программирования, в JavaScript над объектами можно выполнять те же самые действия и с помощью функций.

3.1.1. Функции в качестве объектов высшего порядка

Функции в JavaScript обладают всеми возможностями объектов, а следовательно, их вполне допустимо трактовать как и любые другие объекты в этом языке. Говорят, что функции являются объектами *высшего порядка*, и поэтому их также можно

- создавать с помощью литералов:

```
function ninjaFunction() {}
```

- присваивать переменным, элементам массива и свойствам других объектов:

```
var ninjaFunction = function() {};   // Присвоить новую функцию переменной
ninjaArray.push(function(){});       // Ввести новую функцию в массив
ninja.data = function(){};           // Присвоить новую функцию свойству другого объекта
```

- передавать в качестве аргументов другим функциям:

```
function call(ninjaFunction){
  ninjaFunction();
}
call(function(){});   // Вновь созданная функция передается в качестве аргумента вызываемой функции
```

- возвращать в качестве значений из других функций:

```
function returnNewNinjaFunction() {
  return function(){};   // Возвратить новую функцию
}
```

- наделять свойствами, которые можно динамически создавать и присваивать им значения:

```
var ninjaFunction = function(){};
ninjaFunction.ninja = "Hanzo";
```

Ввести новое свойство в функцию

Все, что допускается делать с объектами, можно делать и с функциями. Функции являются объектами, наделенными особенностью *вызываться*. В частности, функции можно вызывать для выполнения определенных действий.

Функциональное программирование на JavaScript

Внедрение функций в качестве объектов высшего порядка служит первым шагом в направлении *функционального программирования*, сосредоточенного на решении задач путем составления функций (вместо определения последовательностей шагов, как это обычно делается в более распространенных языках императивного программирования). Функциональное программирование помогает в написании кода, который проще тестировать, расширять и помещать в модули. Но эта тема настолько обширна, что в данной книге ей лишь отдается должное внимание (например, в главе 9). Если же вам интересно более подробно ознакомиться с преимуществами функционального программирования и его принципами, а также их применением в программах на JavaScript, рекомендуется прочитать книгу Если же вам интересно более подробно ознакомиться с преимуществами функционального программирования и его принципами, а также их применением в программах на JavaScript, рекомендуется прочитать книгу Луиса Атенсио *Функциональное программирование на JavaScript: как улучшить код JavaScript-программ* (пер. с англ., изд. "Диалектика", ISBN 978-5-9909445-8-9, 2017 г.).

Одна из особенностей объектов высшего порядка заключается в том, что их можно передавать функциям в качестве аргументов. В отношении функций это означает, что одна функция передается в качестве аргумента другой функции, которая может при последующем выполнении приложения вызвать переданную ей функцию. И это служит характерным примером более общего понятия, известного под названием *функции обратного вызова*. Рассмотрим это важное понятие более подробно.

3.1.2. Функции обратного вызова

Всякий раз, когда создается функция для последующего вызова, будь то из браузера или из другого кода, это означает, что, по существу, подготавливается *обратный вызов*. Своим происхождением этот термин обязан тому факту, что функция устанавливается для последующего обратного ее вызова из какого-нибудь другого кода в подходящий момент выполнения.

Обратные вызовы являются важной составляющей эффективного использования JavaScript, и можно с уверенностью сказать, что вам приходилось не раз пользоваться ими в своем коде, где бы этот код ни применялся, будь то обработка события от щелчка кнопкой мыши, получение данных от сервера или анимация отдельных частей пользовательского интерфейса приложения.

В этом разделе будет показано, как пользоваться обратными вызовами для обработки событий или упрощения сортировки коллекций — типичных примеров применения обратных вызовов. Это несколько более сложные примеры, и поэтому разберем полностью понятие обратного вызова в его простейшей форме. Начнем со следующего совершенно бесполезного, но наглядного примера одной функции, которой в качестве параметра передается ссылка на другую функцию для ее последующего вызова:

```
function useless(ninjaCallback) {
  return ninjaCallback();
}
```

Какой бы бесполезной эта функция ни оказалась, она все же помогает ясно понять, каким образом одна функция сначала передается другой в качестве аргумента, а затем вызывается с помощью переданного параметра. Рассматриваемую здесь бесполезную функцию можно протестировать с помощью кода из листинга 3.1.

Листинг 3.1. Простой пример обратного вызова

```
var text = "Domo arigato!";
report("Before defining functions");

function useless(ninjaCallback) {
  report("In useless function");
  return ninjaCallback();
}

function getText() {
  report("In getText function");
  return text;
}

report("Before making all the calls");

assert(useless(getText) === text,
  "The useless function works! " + text);

report("After the calls have been made");
```

Определить функцию, которой передается и сразу вызывается функция обратного вызова

Определить простую функцию, возвращающую значение глобальной переменной

Вызвать бесполезную функцию и передать ей функцию обратного вызова `getText()`

В примере кода из листинга 3.1 применяется специальная функция `report()` для вывода нескольких сообщений, позволяющих отслеживать ход выполнения данного кода. Эта функция описана в приложении Б. Кроме того, в данном примере кода применяется функция `assert()`, упоминавшаяся в главе 1. Обычно этой функции передается два аргумента. В качестве первого ее аргумента указывается выражение, исходное условие которого утверждается. В данном случае требуется выяснить, возвращается ли в результате вызова функции `useless()` вместе с функцией `getText()` в качестве ее аргумента значение переменной `text` (утверждение `useless(getText) === text`). Если вычисляется логическое значение `true` первого аргумента функции `assert()`, то данное утвержде-

ние проходит, а иначе оно не проходит. В качестве второго аргумента функции `assert()` указывается сообщение, которое связано с данным утверждением и обычно выводится вместе с соответствующим индикатором прохождения или непрохождения. (Тестирование в целом и наша собственная скромная реализация функций `assert()` и `report()` рассматривается в приложении Б.)

Результат выполнения рассматриваемого здесь примера кода приведен на рис. 3.1. Как видите, если вызвать функцию `useless()` с функцией `getText()` в качестве ее аргумента, то возвратится предполагаемое значение.

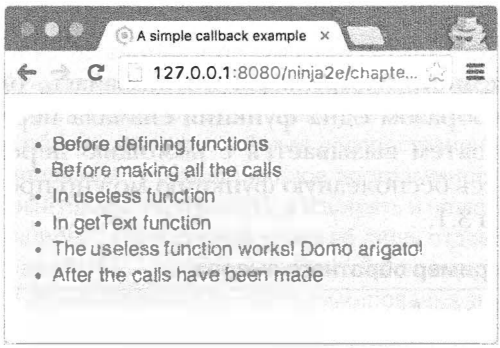


Рис. 3.1. Результат выполнения кода из листинга 3.1

Проанализируем подробно, каким образом выполняется столь простой обратный вызов в данном примере кода. Как показано на рис. 3.2, функция `getText()` передается функции `useless()` в качестве ее аргумента. Это означает, что в теле функции `useless()` обращение к функции `getText()` может происходить через параметр `ninjaCallback`. Таким образом, обратный вызов функции `ninjaCallback()` приводит к выполнению функции `getText()`. В частности, функция `getText()`, переданная в качестве аргумента функции `useless()`, вызывается из нее обратно.

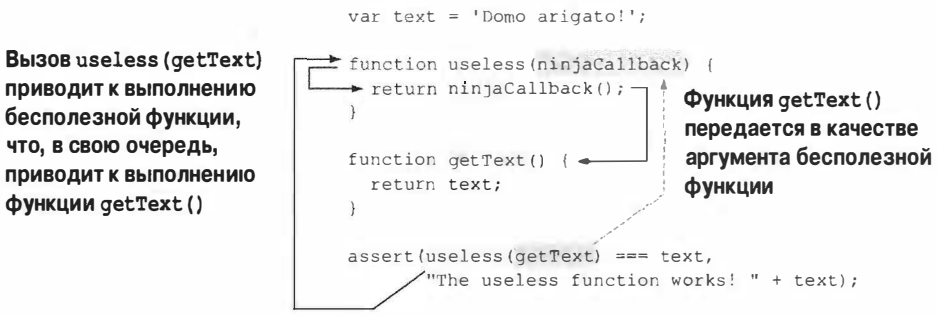


Рис. 3.2. Порядок выполнения кода при вызове функции **`useless(getText)`**. Функция **`useless()`** вызывается вместе с функцией **`getText()`** в качестве ее аргумента. В теле функции **`useless()`** вызывается переданная ей функция, что в данном случае приводит к выполнению функции **`getText()`** (т.е. к ее обратному вызову)

Обратный вызов организуется так просто благодаря функциональному характеру языка JavaScript, позволяющему обращаться с функциями как с объектами высшего порядка. Можно даже пойти еще дальше, переписав рассматриваемый здесь пример кода следующим образом:

```
var text = 'Domo arigato!';
function useless(ninjaCallback) {
  return ninjaCallback();
}

assert(useless(function () { return text;}) === text,
       "The useless function works! " + text);
```

Определить функцию обратного вызова прямо в аргументе

Одна из самых примечательных особенностей языка JavaScript заключается в возможности создавать функции везде, где только может появиться выражение в коде. Это не только делает код более компактным и простым для понимания (благодаря тому что определения функций оказываются рядом с местами их применения), но и позволяет исключить засорение глобального пространства имен излишними именами, если не предполагается обращаться к функции из разных мест в прикладном коде.

В предыдущем примере обратного вызова мы сами сделали обратный вызов. Но обратные вызовы может делать и браузер. Вернемся к примеру веб-приложения из главы 2, где имеется следующий фрагмент кода:

```
document.body.addEventListener("mousemove", function() {
  var second = document.getElementById("second");
  addMessage(second, "Event: mousemove");
});
```

Здесь также указана функция обратного вызова. Она определяется в качестве обработчика событий типа `mousemove` и вызывается браузером при наступлении такого события.

Примечание

В этом разделе обратные вызовы представлены в качестве функций, которые обратно вызываются в другом коде, когда в дальнейшем наступает подходящий момент. В рассмотренном выше примере кода обратный вызов (функции **`useless()`**) делается немедленно, а в примере веб-приложения из главы 2 — из браузера, когда наступает соответствующее событие **`mousemove`**. Следует особо подчеркнуть, что, в отличие от нас, некоторые считают, что обратный вызов должен делаться асинхронно, и поэтому в первом примере на самом деле обратный вызов не происходит. Мы упоминаем об этом на тот случай, если у вас возникнет с кем-то горячая полемика по поводу обратных вызовов.

Сортировка путем сравнения

Практически всегда, когда имеется коллекция данных, ее нужно каким-то образом отсортировать. Допустим, имеется следующий массив чисел, расположенных в произвольном порядке: **0**, **3**, **2**, **5**, **7**, **4**, **8**, **1**. Такой порядок располо-

жения чисел может оказаться вполне подходящим, но рано или поздно числа, скорее всего, придется отсортировать в каком-нибудь другом порядке.

Как правило, реализация алгоритмов сортировки относится далеко не к самым тривиальным задачам программирования. Для выполнения конкретного задания нужно выбрать наилучший алгоритм, реализовать и адаптировать его к текущим потребностям, чтобы отсортировать элементы в нужном порядке, и при этом постараться не внести программные ошибки. Из всех этих задач конкретный прикладной характер имеет лишь порядок сортировки. Правда, для сортировки всех массивов в JavaScript имеется метод `sort()`, для которого требуется лишь определить алгоритм сортировки значений в заданном порядке.

И здесь на помощь приходят обратные вызовы! Вместо того чтобы просто предоставить алгоритму сортировки возможность самому выяснять, в каком именно порядке должны следовать числовые значения, мы создадим функцию, выполняющую сравнение, обеспечив алгоритму сортировки доступ к ней в форме обратного вызова. Следовательно, она будет вызываться из этого алгоритма всякий раз, когда возникнет потребность в сравнении. Функция обратного вызова должна возвратить положительное значение, если порядок следования переданных ей значений должен быть изменен на обратный; отрицательное значение, если этот порядок не нужно изменять; и нулевое значение, если значения одинаковы. Возвращаемое значение, требующееся для сортировки массива, получается путем вычитания сравниваемых значений, как показано ниже.

```
var values = [0, 3, 2, 5, 7, 4, 8, 1];

values.sort(function(value1, value2){
  return value1 - value2;
});
```

В данном случае отпадает необходимость обдумывать подробности действия алгоритма сортировки на низком уровне или даже выбирать конкретный алгоритм сортировки. Достаточно предоставить функцию обратного вызова, чтобы интерпретатор JavaScript вызывал ее всякий раз, когда требуется сравнить два элемента.

Функциональный подход позволяет создать функцию как самостоятельную сущность и передать ее в качестве аргумента методу, который может ее принять в качестве параметра, подобно любому другому типу объекта. Именно в этом и проявляется примечательная особенность функций как объектов высшего порядка.

3.2. Особенности применения функций в качестве объектов

В этом разделе мы рассмотрим ряд способов, позволяющих выгодно воспользоваться сходством функций с объектами других типов. В частности, ни-

что не мешает нам, как ни странно, добавить свойства к функциям следующим образом:

```
var ninja = {};
ninja.name = "hitsuke";
```

Создать объект и присвоить ему новое свойство

```
var wieldSword = function(){};
wieldSword.swordType = "katana";
```

Создать функцию и присвоить ей новое свойство

Рассмотрим две более любопытные возможности, которые открывает данная особенность функций.

- **Сохранение функций в коллекции**, которое позволяет легко манипулировать связанными вместе функциями. Например, делать обратные вызовы, когда происходит что-нибудь интересное.
- **Запоминание**, которое позволяет запоминать вычисленные ранее значения, повышая тем самым производительность при последующих вызовах.

Рассмотрим эти возможности более подробно.

3.2.1. Сохранение функций

В некоторых случаях, когда, например, требуется манипулировать коллекциями обратных вызовов, вызываемых при наступлении определенных событий, нужно хранить коллекции уникальных функций. Но при вводе функций в такую коллекцию может возникнуть затруднение, связанное с тем, что нужно отделить те функции, которые являются новыми для коллекции и поэтому должны быть в нее введены, от тех функций, которые уже находятся в коллекции и поэтому не должны в нее вводиться. В общем, манипулируя коллекциями обратных вызовов, следует избегать любых дубликатов, чтобы единственное событие не стало причиной многих вызовов одной и той же функции обратного вызова.

Очевидно, хотя и наивно предположить, что все функции можно сначала сохранить в массиве, а затем организовать циклическое обращение к элементам этого массива, чтобы проверить, не дублируются ли функции. К сожалению, это малоэффективный способ, который не годится для мастера, стремящегося к тому, чтобы программа не просто работала, а работала надежно и эффективно. Поэтому для достижения поставленной цели на подходящем уровне сложности можно воспользоваться свойствами функций, как показано в листинге 3.2.

Листинг 3.2. Сохранение уникальных функций в коллекции

Отслеживать следующий доступный для присваивания идентификатор

Создать объект, служащий в качестве кеша для хранения функций

```
var store = {
  nextId: 1,
  cache: {},
```

```
  add: function(fn) {
    if (!fn.id) {
      fn.id = this.nextId++;
      this.cache[fn.id] = fn;
      return true;
    }
  }
};
function ninja(){}
assert(store.add(ninja),
       "Function was safely added.");
assert(!store.add(ninja),
       "But it was only added once.");
```

Поместить функции в кеш, но только в том случае, если они уникальны

Проверить, работает ли все так, как и запланировано

В примере кода из листинга 3.2 сначала создается объект, который присваивается переменной `store` для дальнейшего хранения уникального набора функций. У этого объекта имеются два свойства данных: одно — для хранения следующего доступного значения идентификатора `id`, а другое — для кеширования сохраняемых функций. Функции добавляются в кеш с помощью приведенного ниже метода `add()`.

```
add: function(fn) {
  if (!fn.id) {
    fn.id = this.nextId++;
    this.cache[fn.id] = fn;
    return true;
  }
...
```

В методе `add()` сначала проверяется, находится ли вводимая функция в коллекции, по наличию ее свойства `id`. Если у текущей функции имеется свойство `id`, то предполагается, что функция уже обработана, и поэтому она просто игнорируется. В противном случае свойство `id` назначается функции и попутно инкрементируется значение свойства `nextId`, а сама функция вводится как свойство в объект `cache`, используя значение `id` в качестве имени свойства. Затем возвращается логическое значение `true`, и тем самым после вызова метода `add()` уведомляется, что функция введена в коллекцию.

Если выполнить рассматриваемый здесь код на веб-странице в браузере, то при попытке в ходе тестирования ввести функцию `ninja()` в коллекцию дважды она вводится лишь один раз, как показано на рис. 3.3. В главе 9 будет продемонстрирован еще более совершенный способ обращения с коллекциями уникальных элементов, в котором применяются множества. Это новый тип объектов появился в стандарте ES6.

Еще один полезный прием, который рекомендуется взять на вооружение, используя свойства функции, состоит в том, чтобы предоставить функции возможность видоизменяться. Таким приемом можно было бы воспользоваться для запоминания вычисленных ранее значений, экономя время при последующих вычислениях.

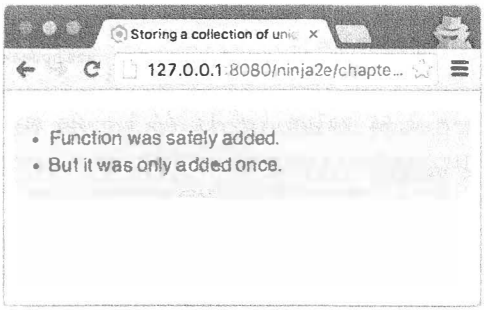


Рис. 3.3. Создав свойство для функции, мы можем отслеживать его значение. Благодаря этому функция будет введена в кеш только один раз

3.2.2. Самозапоминающиеся функции

Как упоминалось ранее, *запоминание* (memoization) представляет собой процесс построения функции, способной запоминать свои вычисленные ранее значения. По существу, всякий раз, когда функция вычисляет результат, он сохраняется вместе с ее аргументами. Это дает возможность возвратить сохраненный ранее результат вместо того, чтобы вычислять его снова при очередном вызове функции с теми же самыми аргументами. Тем самым можно заметно повысить производительность приложения, избегая повторения лишний раз сложных вычислений, которые уже были один раз выполнены. Запоминание особенно удобно при сложных вычислениях для воспроизведения анимации, поиске данных, которые нечасто изменяются, или при выполнении любых длительных математических расчетов.

В качестве элементарного примера рассмотрим простой (и разумеется, не особенно эффективный) алгоритм вычисления простых чисел. И хотя это довольно простой пример, тем не менее он демонстрирует прием, который можно вполне применить в сложных и затратных вычислениях, например, для получения хеш-кода символьной строки по алгоритму шифрования MD5. Такие вычисления слишком сложны, чтобы продемонстрировать их здесь на конкретных примерах.

Внешне самозапоминающаяся функция практически ничем не отличается от любой обычной функции, но в ее теле скрытно создается кеш, в котором она будет сохранять результаты выполняемых вычислений. Исходный код этой функции приведен в листинге 3.3.

Листинг 3.3. Запоминание вычисленных ранее значений

```
function isPrime(value){
  if (!isPrime.answers){
    isPrime.answers = {};
  }
```

Создать кеш

```
  if (isPrime.answers[value] !== undefined) {
    return isPrime.answers[value];
  }

  var prime = value !== 0 && value !== 1; // 1 — не простое число

  for (var i = 2; i < value; i++) {
    if (value % i === 0) {
      prime = false;
      break;
    }
  }
  return isPrime.answers[value] = prime;
}

assert(isPrime(5), "5 is prime!" );
assert(isPrime.answers[5], "The answer was cached!" );
```

Проверить наличие кешированных значений

Сохранить вычисленное значение

Проверить, все ли работает нормально

Сначала в функции `isPrime()` проверяется, создано ли свойство `answers`, которое предполагается использовать в качестве кеша. И если это свойство отсутствует, то оно создается, как показано ниже.

```
if (!isPrime.answers) {
  isPrime.answers = {};
}
```

Создание пустого кеша происходит лишь при первом вызове функции, после чего он уже существует. Затем проверяется, запомнен ли в кеше `answers` результат вычисления переданного значения:

```
if (isPrime.answers[value] !== undefined) {
  return isPrime.answers[value];
}
```

Для сохранения результатов вычислений в виде логических значений (`true` или `false`) в этом кеше используется значение аргумента функции в качестве имени (ключа) свойства. Если ответ найден в кеше, он просто возвращается.

Если же в кеше не найдено ни одного значения, то выполняются вычисления с целью определить, является ли значение простым числом (для больших значений такая операция может оказаться весьма затратной), а результат сохраняется в кеше при возврате из функции, как показано ниже.

```
return isPrime.answers[value] = prime;
```

Кеш является свойством самой функции, и поэтому он действует до тех пор, пока действует сама функция. И, наконец, в рассматриваемом здесь примере кода работоспособность запоминания проверяется следующим образом:

```
assert(isPrime(5), "5 is prime!" );
assert(isPrime.answers[5], "The answer was cached!" );
```

У такого подхода имеются два главных преимущества.

- Конечный пользователь запрашивает вычисленное ранее значение, выгодно используя преимущества вызова функции.
- Механизм запоминания действует слаженно и незаметно. Ни конечному пользователю, ни автору веб-страницы не нужно делать какие-либо специальные запросы или другие лишние инициализирующие операции, чтобы привести этот механизм в действие.

Но у данного подхода имеются также свои недостатки, которые могут даже перевесить его преимущества.

- Любого рода кеширование, безусловно, приводит к излишнему расходу оперативной памяти с целью повышения производительности.
- Пуристы могут заявить, что кеширование не следует смешивать с бизнес-логикой, поскольку функция или метод должны делать что-нибудь одно, и делать это исправно. Впрочем, в главе 8 будет показано, как опровергнуть это заявление.
- Проверить в реальных условиях или измерить производительность такого алгоритма нелегко, поскольку результаты его работы будут зависеть от предыдущих входных данных функции.

Итак, рассмотрев некоторые практические примеры применения функций в качестве объектов высшего порядка, перейдем к исследованию различных способов определения функций.

3.3. Определение функций

Функции в JavaScript обычно определяются с помощью *функционального литерала*, создающего значение функции таким же образом, как, например, числовой литерал создает числовое значение. Напомним, что как объекты высшего порядка функции являются значениями, которыми можно пользоваться, как и любыми другими значениями в JavaScript, например, строковыми или числовыми. И вам, скорее всего, приходилось делать это не раз, осознанно или не осознанно.

В языке JavaScript предоставляется ряд способов определения функций, которые можно разделить на четыре группы.

- **Объявления функций и функциональные выражения.** Это два наиболее распространенных, хотя и несколько отличающихся способа определения функций. Зачастую их даже не разделяют, но, как станет ясно в дальнейшем, ясное представление об их отличиях помогает лучше понять, когда именно функции доступны для вызова. Ниже приведен пример объявления функции.

```
function myFun(){ return 1;}
```

- **Стрелочные функции**, часто называемые **лямбда-функциями.** Этот тип функций лишь недавно появился в стандарте ES6 языка JavaScript. Он

позволяет определить функции с помощью более простого синтаксиса и даже разрешить одно типичное затруднение, возникающее в связи с функциями обратного вызова и рассматриваемое далее. Ниже приведен пример определения стрелочной функции.

```
myArg => myArg*2
```

- **Конструкторы функций.** Это нечасто применяемый способ определения функций, позволяющий динамически конструировать новую функцию из символьной строки, которая также может быть сформирована динамически. В приведенном ниже примере динамически создается функция, принимающая два параметра, `a` и `b`, и возвращающая сумму значений этих параметров.

```
new Function('a', 'b', 'return a + b')
```

- **Функции-генераторы.** Этот тип функций также был введен в стандарт ES6 языка JavaScript. Он позволяет создать функцию, из которой, в отличие от обычных функций, можно выйти и снова войти в нее при последующем выполнении приложения, сохраняя значения ее переменных в промежутках между последовательными выходами и возвратами к данной функции. Определить можно генераторные варианты объявлений функций, функциональных выражений и конструкторов функций. Ниже приведен пример определения функции-генератора.

```
function* myGen(){ yield 1; }
```

Очень важно понимать отличия перечисленных выше способов, поскольку от выбранного способа определения функции в значительной степени зависит, когда функция доступна для вызова, как она себя ведет и для какого объекта может быть вызвана.

В этой главе рассматриваются объявления функций, функциональные выражения и стрелочные функции. Ниже поясняется их принцип действия, но нам еще не раз придется возвращаться к ним в данной книге для исследования их особенностей. С другой стороны, функции-генераторы довольно специфичны и существенно отличаются от обычных функций, и поэтому они отдельно рассматриваются в главе 6.

Что же касается конструкторов функций, то мы вообще не будем рассматривать их в данной книге. И хотя они находят ряд интересных применений, особенно при динамическом создании и интерпретации кода, мы считаем их тупиковым языковым средством JavaScript. Если же вы желаете поближе ознакомиться с конструкторами функций, обратитесь к документации, доступной по адресу `https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function`.

Итак, начнем с самых простых и традиционных способов определения функций: объявлений функций и функциональных выражений.

3.3.1. Объявления функций и функциональные выражения

Двумя наиболее распространенными способами определения функций в JavaScript являются объявления функций и функциональные выражения. Эти два способа настолько сходны, что нередко вообще не различаются. Но, как поясняется в последующих главах, эти отличия все же существуют, хотя они едва заметны.

Объявления функций

Самый простой способ определить функцию в JavaScript – воспользоваться объявлением функции (рис. 3.4). Как видите, всякое объявление функции начинается с обязательного ключевого слова `function`, после которого следует обязательное имя функции и список имен необязательных параметров, указываемых через запятую и заключаемых в обязательные круглые скобки. Тело функции, которое потенциально является пустым списком операторов, должно быть заключено в фигурные скобки. Помимо этой формы, которой должно удовлетворять всякое объявление функции, имеется еще одно условие: объявление функции должно занимать самостоятельное положение как отдельный оператор JavaScript, хотя оно может содержаться в теле другой функции или блока кода. В следующем разделе будет показано, что именно мы имели в виду.

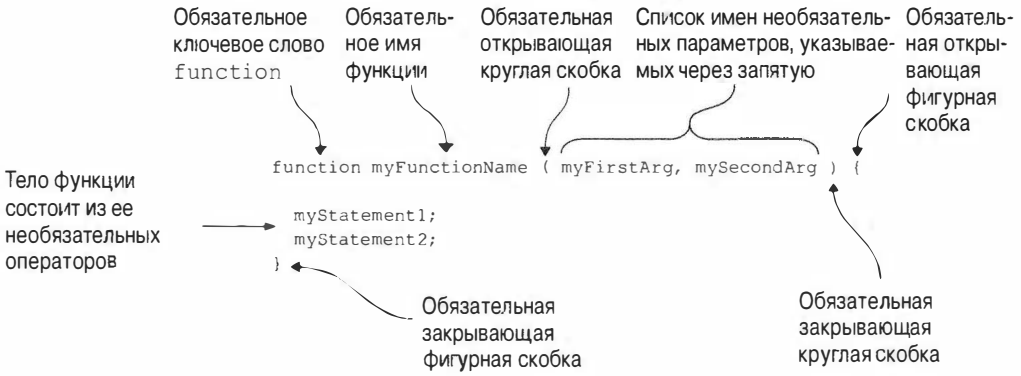


Рис. 3.4. Объявление функции занимает самостоятельное положение в отдельном блоке кода JavaScript! Хотя оно может содержаться и в теле других функций

В листинге 3.4 приведены два наглядных примера объявления функций.

Листинг 3.4. Примеры объявления функций

```
function samurai() {
  return "samurai here";
}

function ninja() {

  function hiddenNinja() {
  return "ninja here";
  }
```

Определить функцию `samurai()` в глобальном коде

Определить функцию `ninja()` в глобальном коде

Определить функцию `hiddenNinja()` в теле функции `ninja()`

```
  return hiddenNinja();
}
```

Внимательно приглядевшись к коду из листинга 3.4, вы можете найти его непривычным, если у вас недостаточно опыта программирования на функциональных языках. Подумать только: одна функция определяется в теле другой функции!

```
function ninja() {
  function hiddenNinja() {
    return "ninja here";
  }
  return hiddenNinja();
}
```

В языке JavaScript это совершенно нормальное явление. И этот пример приведен здесь специально, чтобы лишний раз подчеркнуть особое значение функций в JavaScript.

Примечание

Наличие одних функций в теле других может вызвать немало трудных вопросов, касающихся области видимости и соответствия имен идентификаторов. Но отложим их на время, поскольку мы еще вернемся к данному примеру кода, рассмотрев его подробнее в главе 5.

Функциональные выражения

Как неоднократно упоминалось ранее, функции в JavaScript являются объектами высшего порядка, а это, среди прочего, означает, что их можно создавать с помощью литералов, присваивать переменным и свойствам и передавать другим функциям в качестве аргументов и возвращать из них в качестве значений. Вследствие того что функции являются основополагающими языковыми конструкциями, в JavaScript их можно трактовать таким же образом, как и любые другие выражения. Таким образом, подобно числовым литералам, как в следующем примере кода:

```
var a = 3;
myFunction(4);
```

в том же самом месте кода можно воспользоваться функциональными литералами следующим образом:

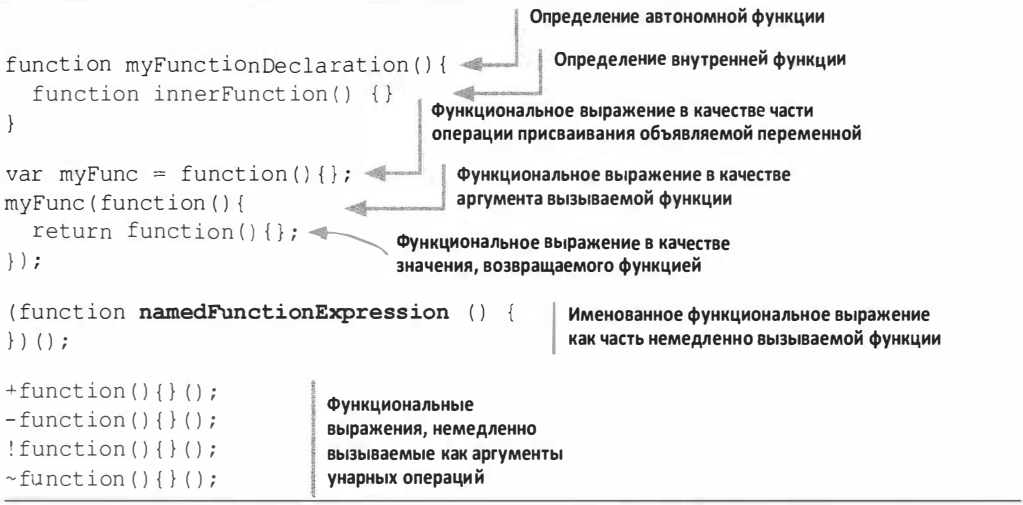
```
var a = function() {};
myFunction(function(){});
```

Подобные функции, которые всегда являются составной частью другого оператора (например, они указываются в правой части выражения присваивания или в качестве аргумента другой функции), называются *функциональными выражениям*. Такие выражения примечательны тем, что они позволяют опре-

делить функции именно там, где они больше всего нужны, тем самым упрощая понимание кода.

В листинге 3.5 демонстрируются отличия объявлений функций от функциональных выражений.

Листинг 3.5. Объявления функций и функциональные выражения




```
function myFunctionDeclaration(){
  function innerFunction() {}
}

var myFunc = function(){};
myFunc(function(){
  return function(){};
});

(function namedFunctionExpression () {
})();

+function(){}();
-function(){}();
!function(){}();
~function(){}();
```

Данный пример кода начинается со стандартного объявления функции, содержащего еще одно объявление внутренней функции, как показано ниже. В данном случае объявления функций представлены отдельными операторами JavaScript, но они могут содержаться и в теле других функций.

```
function myFunctionDeclaration(){
  function innerFunction() {}
}
```

С другой стороны, функциональные выражения всегда являются частью какого-нибудь другого оператора. Они указываются на уровне выражения, как, например, в правой части объявления переменной (или операции присваивания):

```
var myFunc = function(){};
```

Кроме того, функциональные выражения могут быть указаны в качестве аргумента вызываемой функции или значения, возвращаемого функцией:

```
myFunc(function() {
  return function(){};
});
```

Помимо местоположения в коде, имеется еще одно отличие объявлений функций от функциональных выражений. Если наличие имени функции в объявлениях функций *обязательно*, то в функциональных выражениях оно совсем *необязательно*.

Имя функции должно быть непременно указано в объявлениях функций потому, что они занимают самостоятельное положение. Одно из основных требований к функции состоит в том, что она должна быть вызываемой, и поэтому к ней нужно каким-то образом обращаться, а это можно сделать только по ее имени.

А функциональные выражения являются частью других выражений в JavaScript, предоставляя альтернативные способы вызова функций. Так, если функциональное выражение присваивается переменной, функцию можно вызвать с помощью этой переменной, как показано ниже.

```
var doNothing = function(){};
doNothing();
```

Если же функциональное выражение указывается в качестве аргумента другой функции, то в ее теле можно вызвать данное выражение по совпадающему имени параметра, как показано в следующем примере кода:

```
function doSomething(action) {
  action();
}
```

Немедленно вызываемые функции

Функциональные выражения можно размещать даже там, где их местонахождение может показаться не вполне уместным, например, там, где обычно предполагается идентификатор функции. Рассмотрим подробнее такую языковую конструкцию (рис. 3.5).

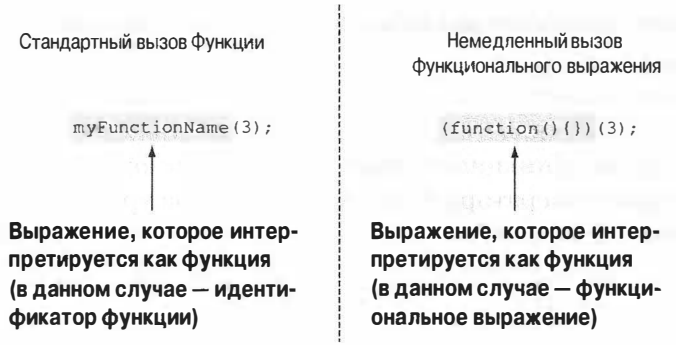


Рис. 3.5. Стандартный вызов функции в сравнении с немедленным вызовом функционального выражения

Если требуется вызвать функцию, то с этой целью используется выражение, интерпретируемое как функция с последующей парой круглых скобок для ее вызова, в которых могут содержаться аргументы. В самом простом случае при вызове функции указывается идентификатор, который интерпретируется как функция (см. рис. 3.5, *слева*). Но выражение, находящееся слева от круглых скобок вызова функции, совсем не обязательно должно быть идентификатором.

Это может быть *любое* выражение, интерпретируемое как функция. Чтобы, например, указать выражение, которое интерпретируется как функция, проще всего воспользоваться функциональным выражением. Как показано на рис. 3.5, *справа*, сначала создается функция, а затем она немедленно вызывается. И такая языковая конструкция называется *немедленно вызываемым функциональным выражением* (IIFE), а короче — *немедленно вызываемой функцией*. Немедленно вызываемые функциональные выражения играют очень важную роль в разработке приложений на JavaScript, поскольку позволяют имитировать модули в JavaScript. Более подробно их применение рассматривается в главе 11.

О круглых скобках, в которые заключается функциональное выражение

В конструкции немедленно вызываемого выражения может вызвать недоумение наличие круглых скобок, в которые заключается это выражение. Зачем они вообще нужны? Они нужны исключительно из синтаксических соображений. Синтаксический анализатор кода JavaScript должен быть в состоянии легко отличать объявления функций от функциональных выражений. Если не заключить функциональное выражение в круглые скобки и указать немедленный вызов в отдельном операторе, например `function(){}(3)`, то синтаксический анализатор кода JavaScript обработает его и придет к выводу, что это объявление функции, поскольку оно указано в отдельном операторе, начинающемся с ключевого слова `function`. Но поскольку всякое объявление функции должно непременно содержать имя функции, а в данном случае оно отсутствует, то будет выдана ошибка. Во избежание этого функциональное выражение заключается в круглые скобки, указывающие синтаксическому анализатору кода JavaScript на то, что это выражение, а не оператор.

Имеется другой, более простой, хотя и, как ни странно, реже применяемый способ достичь той же самой цели: `(function(){}(3))`. Заключив определение и вызов немедленно вызываемой функции в круглые скобки, можно также указать синтаксическому анализатору кода JavaScript на то, что это выражение.

Четыре последних выражения в примере кода из листинга 3.5 являются вариациями на ту же самую тему немедленно вызываемых функциональных выражений. Такие выражения нередко встречаются в библиотеках JavaScript.

```
+function(){}();
-function(){}();
!function(){}();
~function(){}();
```

На этот раз вместо заключения функциональных выражений в круглые скобки с целью отличить их от объявлений функций используются унарные операции `+`, `-`, `!`, `~`. Подобным способом интерпретатору JavaScript указывается на то, что это выражения, а не операторы. Следует, однако, иметь в виду, что результаты применения этих унарных операций нигде не сохраняются. С точки зрения вычислений значение имеют не они, а вызовы немедленно вызываемых функциональных выражений.

Итак, рассмотрев особенности двух самых основных способов определения функций в JavaScript (объявлений функций и функциональных выражений), перейдем к исследованию *стрелочных функций* — нововведения в стандарте ES6 языка JavaScript.

3.3.2. Стрелочные функции

Примечание

Стрелочные функции были введены в стандарт ES6 языка JavaScript (подробнее о совместимости с браузерами см. таблицу по адресу **http://kangax.github.io/compat-table/es6/#test-arrow_functions**.)

В приложениях на JavaScript применяется немало функций, и поэтому имело смысл ввести некоторое "синтаксическое удобство", которое позволяло бы создавать функции более кратким, лаконичным способом, упростив жизнь разработчикам.

Стрелочные функции во многом упрощают функциональные выражения. Вернемся к примеру сортировки коллекций из раздела 3.1.2, рассмотрев следующий фрагмент кода:

```
var values = [0, 3, 2, 5, 7, 4, 8, 1];
values.sort(function(value1,value2){
  return value1 - value2;
});
```

В данном примере применяется функциональное выражение обратного вызова, указываемое при вызове метода сортировки для объекта массива. Этот обратный вызов делается интерпретатором JavaScript для сортировки значений массива по убывающей.

А ниже показано, как достичь той же самой цели с помощью стрелочных функций. Обратите внимание, насколько более лаконичным получается код.

```
var values = [0, 3, 2, 5, 7, 4, 8, 1];
values.sort((value1,value2) => value1 - value2);
```

В нем отсутствует нагромождение из ключевого слова `function`, фигурных скобок и оператора `return`. Стрелочная функция намного проще, чем функциональное выражение, определяет функцию, которой передается два аргумента и возвращает их разность. Обратите внимание на новую операцию **=>**, называемую *"толстой стрелкой"*, обозначаемую знаками равенства и "больше" и служащую основанием для определения стрелочной функции.

А теперь разберем по частям синтаксис стрелочной функции, начиная со следующего простейшего способа ее определения:

```
param => expression
```

Этой стрелочной функции передается параметр, а она возвращает значение выражения. Пример применения такого синтаксиса стрелочных функций демонстрируется в листинге 3.6.

Листинг 3.6. Стрелочная функция в сравнении с функциональным выражением

```
var greet = name => "Greetings " + name;
assert(greet("Oishi") ===
       "Greetings Oishi", "Oishi is properly greeted");

var anotherGreet = function(name){
  return "Greetings " + name;
};
assert(anotherGreet("Oishi") === "Greetings Oishi",
       "Again, Oishi is properly greeted");
```

Определить стрелочную функцию

Определить функциональное выражение

Оцените, насколько более компактным делают код стрелочные функции, не принося в жертву его ясность. Это простейшая форма синтаксиса стрелочных функций, а в общем, стрелочные функции могут быть определены двумя способами, как показано на рис. 3.6.

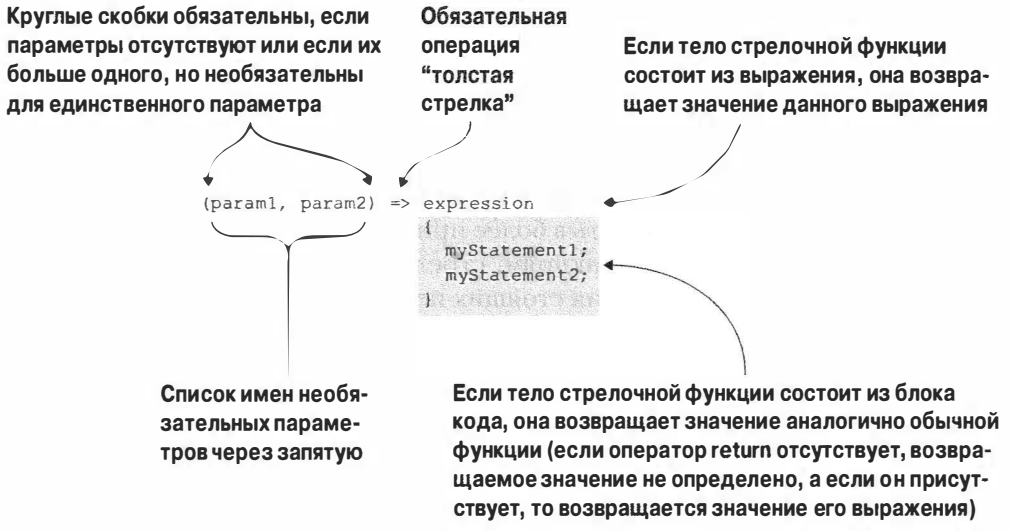


Рис. 3.6. Синтаксис стрелочных функций

Как видите, определение стрелочной функции начинается со списка имен необязательных параметров, указываемых через запятую. Если параметры отсутствуют или их больше одного, то их список должен быть заключен в круглые скобки. Но если указывается единственный параметр и следующая за ним операция "толстая стрелка", то круглые скобки необязательны. После списка параметров следует операция "толстая стрелка", которая указывает интерпретатору JavaScript на то, что это стрелочная функция.

После операции "толстая стрелка" можно указать два возможных варианта. Если определяется простая функция, то в данном месте указывается выражение, состоящее из математической операции, вызова другой функции и прочего, а в результате вызова стрелочной функции будет возвращено значение данного выражения. Так, в примере кода из листинга 3.6 определена следующая стрелочная функция:

```
var greet = name => "Greetings " + name;
```

Значение, возвращаемое этой функцией, представляет собой результат сцепления символьной строки `"Greetings "` со значением параметра `name`.

В других случаях, когда определяются сложные стрелочные функции, требующие больше кода, после операции "толстая стрелка" можно указать блок кода:

```
var greet = name => {
  var helloString = 'Greetings ';
  return helloString + name;
};
```

В данном случае стрелочная функция возвращает значение таким же образом, как и обычная функция. Если оператор `return` отсутствует, то результат вызова стрелочной функции оказывается неопределенным (`undefined`). А если этот оператор присутствует, то в результате вызова стрелочной функции возвратится значение выражения, вычисляемого в данном операторе.

Мы еще не раз вернемся к стрелочным функциям на протяжении всей данной книги. Среди прочего, мы представим дополнительные возможности стрелочных функций, которые позволяют избежать едва заметных программных ошибок, которые могут возникать в более привычных функциях.

Подобно всем остальным функциям, стрелочным функциям могут передаваться аргументы для выполнения стоящих перед ними задач. Выясним далее, что же происходит со значениями, которые передаются функции.

3.4. Аргументы и параметры функций

При обсуждении функций термины *аргумент* и *параметр* нередко употребляются попеременно, как будто они более или менее обозначают одно и то же. Но теперь придется поступить более формально, определив их следующим образом.

- *Параметр* – это переменная, которая указывается как часть определения функции.
- *Аргумент* – это значение, которое передается функции при ее вызове.

Отличие аргументов от параметров наглядно показано на рис. 3.7.

Как видите, параметр функции указывается при ее определении. Параметры имеются у всех типов определяемых функций.

- Объявлений функций (параметр `ninja` функции `skulk()`).

- Функциональных выражений (параметры `person` и `action` функции `performAction()`).
- Стрелочных функций (параметр `daimyo`).

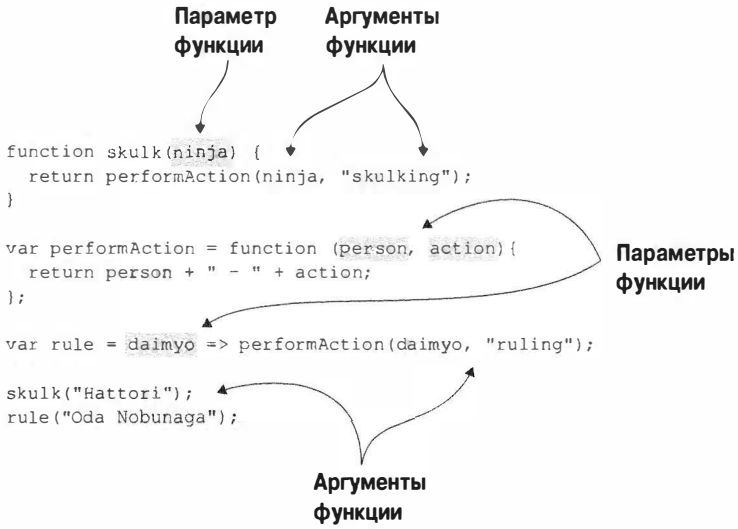


Рис. 3.7. Отличие аргументов функций от их параметров

С другой стороны, аргументы связаны непосредственно с вызовом функции. Они представляют собой значения, передаваемые функции в момент ее вызова.

- Символьная строка `"Hattori"` передается в качестве аргумента функции `skulk()`.
- Символьная строка `"Oda Nobunaga"` передается в качестве аргумента функции `rule()`.
- Параметр `ninja` функции `skulk()` передается в качестве аргумента функции `performAction()`.

Когда в вызове функции предоставляется список аргументов, эти аргументы присваиваются параметрам, задаваемым в определении функции в том порядке, в каком они указаны: первый аргумент присваивается первому параметру, второй аргумент — второму и т.д.

Если же число аргументов отличается от числа параметров, то никакой ошибки нет. В языке JavaScript подобная ситуация разрешается следующим образом: если в вызове функции предоставляется больше аргументов, чем задано параметров, то лишние аргументы просто не присваиваются именам параметров, как наглядно показано в примере, приведенном на рис. 3.8.

Как показано на рис. 3.8, если вызвать функцию `practice("Yoshi", "sword", "shadow sword", "katana")`, то аргументы `"Yoshi"`, `"sword"` и `"shadow sword"` будут присвоены параметрам `ninja`, `weapon` и `technique` со-

ответственно. Аргумент "katana" оказывается лишним и поэтому не присваивается ни одному из параметров данной функции. Как поясняется в следующей главе, некоторые аргументы по-прежнему остаются доступными, даже если они не присваиваются именам параметров.

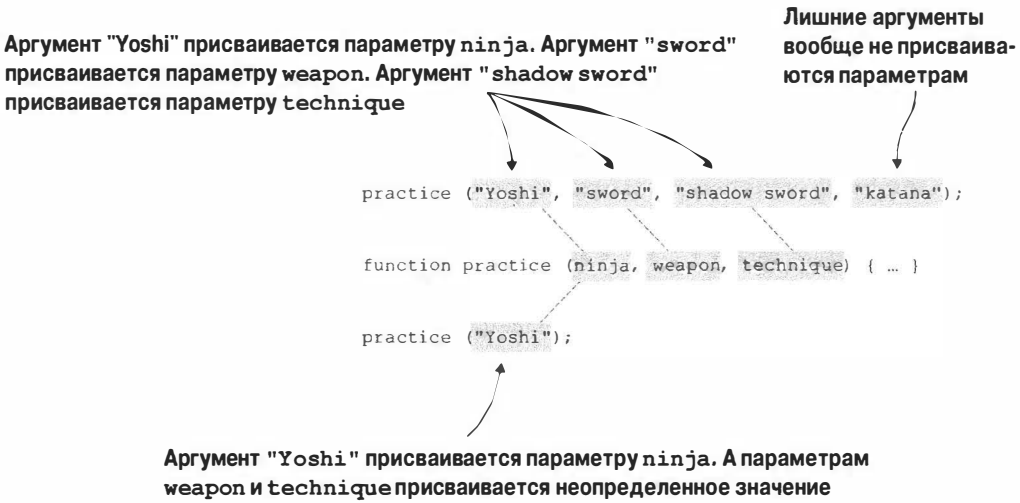


Рис. 3.8. Аргументы присваиваются параметрам функции в указанном порядке. А лишние аргументы не присваиваются ни одному из параметров

Если же у функции имеется больше параметров, чем передано ей аргументов, то параметрам без соответствующих аргументов присваивается неопределенное значение undefined. Так, если сделать вызов practice("Yoshi"), то параметру ninja будет присвоено значение "Yoshi", тогда как параметрам weapon и technique — значение undefined.

Обращение с аргументами и параметрами функции имеет такую же давнюю историю, как и сам язык JavaScript. Поэтому исследуем теперь два новых языковых средства JavaScript, появившихся в стандарте ES6: оставшиеся параметры и стандартные (устанавливаемые по умолчанию) значения параметров.

3.4.1. Оставшиеся параметры

Примечание

Оставшиеся параметры были внедрены в стандарте ES6 языка Java Script (подробнее о совместимости с браузерами см. список по адресу **http://kangax.github.io/compat-table/es6/#test-rest_parameters**.)

В качестве следующего примера рассмотрим функцию, умножающую первый аргумент на самый большой из оставшихся аргументов. И хотя такая функция не находит особого применения на практике, она служит наглядным при-

мером применения дополнительных приемов обращения с аргументами в теле функции.

Казалось бы, что может быть проще: взять первый аргумент и умножить на самый большой из оставшихся аргументов. Но в прежних версиях JavaScript для этого приходилось специально применять обходные приемы, которые будут рассмотрены в следующей главе. Правда, начиная со стандарта ES6, эти приемы больше не потребуются, поскольку можно воспользоваться *оставшимися параметрами*, как демонстрируется в примере кода из листинга 3.7.

Листинг 3.7. Применение оставшихся параметров

```
function multiMax(first, ...remainingNumbers){
  var sorted = remainingNumbers.sort(function(a, b){
    return b - a;
  });
  return first * sorted[0];
}
assert(multiMax(3, 1, 2, 3) == 9,
       "3*3=9 (First arg, by largest.)");
```

Оставшиеся параметры обозначаются префиксом ...

Отсортировать оставшиеся числа по убывающей

Данная функция вызывается таким же образом, как и любая другая

Если обозначить последний параметр префиксом многоточия (...), как показано ниже, он превратится в массив *оставшихся параметров*, содержащий все остальные аргументы, переданные функции.

```
function multiMax(first, ...remainingNumbers){
...
}
```

В данном примере функция `multiMax()` вызывается с четырьмя аргументами: `multiMax(3, 1, 2, 3)`. В теле этой функции значение **3** первого аргумента присваивается ее первому параметру `first`. А поскольку второй параметр данной функции оказывается оставшимся, то все остальные аргументы (**1**, **2**, **3**) размещаются в новом массиве `remainingNumbers`. Затем в данной функции получается наибольшее число путем сортировки массива по убывающей (обратите внимание, насколько просто изменить порядок сортировки) и выбора самого большого числа, находящегося на месте первого элемента отсортированного массива. (Это далеко не самый эффективный способ определения наибольшего числа, но почему бы не воспользоваться с выгодой уже приобретенными навыками программирования на JavaScript?)

Примечание

Оставшимся может быть лишь последний параметр функции. А попытка обозначить префиксом многоточия любой другой параметр функции, кроме последнего, лишь приведет к появлению ошибки в следующей форме:

`SyntaxError: parameter after rest parameter`

(Синтаксическая ошибка: наличие параметра после оставшегося параметра)

В следующем разделе мы продолжим расширение арсенала языковых средств JavaScript, рассмотрев еще одно нововведение ES6: стандартные параметры.

3.4.2. Стандартные параметры

Примечание

Стандартные параметры функций были введены в версии ES6 языка JavaScript (подробнее о совместимости с браузерами см. таблицу по адресу `http://kangax.github.io/compat-table/es6/#test-default_function_parameters`.)

Многие компоненты пользовательского интерфейса веб-приложений (особенно модули, подключаемые из библиотеки jQuery) допускают настройку. Так, если разрабатывается компонент ползунка, пользователям может быть предоставлена возможность указать время срабатывания таймера, по истечении которого один элемент заменяется другим, а также вид анимации, воспроизводимой в течение такой замены. В то же время некоторых пользователей такая возможность может вообще не заинтересовать, поскольку их вполне устраивают исходные настройки. Для таких случаев идеально подходят стандартные параметры!

Упомянутый выше небольшой пример настройки компонента ползунка наглядно показывает лишь конкретную ситуацию, где *почти* во всех вызовах функции используется одно и то же значение отдельного параметра. Поэтому рассмотрим более простой случай, когда почти все наши ниндзя прячутся, кроме Ягыо, которому достаточно действовать тайком, как показано в следующем примере кода:

```
function performAction(ninja, action) {
  return ninja + " " + action;
}
performAction("Fuma", "skulking");
performAction("Yoshi", "skulking");
performAction("Hattori", "skulking");
performAction("Yagyu", "sneaking");
```

Не кажется ли вам не совсем удобным постоянно повторять один и тот же аргумент `skulking` (скрыться) только потому, что Ягыо своевольничает, отказываясь действовать, как подобает настоящему ниндзя?

В других языках программирования это затруднение чаще всего разрешается перегрузкой функций (т.е. определением дополнительных функций с теми же самыми именами параметров, но разными их значениями). К сожалению, в JavaScript перегрузка функций не поддерживается, и поэтому в прошлом разработчики нередко прибегали в подобных случаях к приему, аналогичному приведенному в примере кода из листинга 3.8.

Листинг 3.8. Обращение со стандартными параметрами до ES6

```
function performAction(ninja, action){
  action = typeof action === "undefined" ? "skulking" : action;
  return ninja + " " + action;
}

assert(performAction("Fuma") === "Fuma skulking",
       "The default value is used for Fuma");

assert(performAction("Yoshi") === "Yoshi skulking",
       "The default value is used for Yoshi");

assert(performAction("Hattori") === "Hattori skulking",
       "The default value is used for Hattori");

assert(performAction("Yagyu", "sneaking") === "Yagyu sneaking",
       "Yagyu can do whatever he pleases, even sneak!");
```

Если параметр `action` не определен, использовать его стандартное значение `skulking`, а если он определен, сохранить его переданное значение

Второй аргумент, определяющий значение параметра `action`, не передан; после выполнения первой функции в ее теле по умолчанию установится значение `skulking`

Передать символьную строку в качестве значения параметра `action`; это значение будет использоваться в теле функции

В данном примере определяется функция `performAction()`, в которой с помощью операции `typeof` проверяется, является ли значение параметра `action` неопределенным (`undefined`). И если это именно так, то в данной функции устанавливается значение `skulking` параметра `action`. Если же параметр `action` передается через вызов функции (и не является неопределенным), то его значение сохраняется.

Примечание

В результате выполнения операции **`typeof`** возвращается символьная строка, обозначающая тип операнда. Если же операнд не определен (например, если не предоставлен соответствующий аргумент для параметра функции), то возвращается символьная строка **`"undefined"`**.

Это типичный пример неудобств, возникающих при обращении с параметрами функций. Поэтому в ES6 была введена поддержка *стандартных параметров*, как демонстрируется в примере кода из листинга 3.9.

Листинг 3.9. Обращение со стандартными параметрами в ES6

```
function performAction(ninja, action = "skulking"){
  return ninja + " " + action;
}
```

В стандарте ES6 допускается присваивать значение параметру функции

```
assert(performAction("Fuma") === "Fuma skulking",
       "The default value is used for Fuma");

assert(performAction("Yoshi") === "Yoshi skulking",
       "The default value is used for Yoshi");

assert(performAction("Hattori") === "Hattori skulking",
       "The default value is used for Hattori");
```

Если значение параметра не передано, то используется его стандартное значение

```
assert(performAction("Yagyu", "sneaking") === "Yagyu sneaking",
      "Yagyu can do whatever he pleases, even sneak!");
```

Используется переданное значение параметра

В данном примере кода демонстрируется синтаксис установки в JavaScript стандартных значений параметров функции. Для этого достаточно присвоить стандартное значение соответствующему параметру функции, как выделено ниже полужирным шрифтом.

```
function performAction(ninja, action = "skulking"){
  return ninja + " " + action;
}
```

А затем, когда вызывается функция и значение соответствующего аргумента опускается, как, например, в первых трех приведенных ниже случаях вызова данной функции, используется его стандартное значение (в данном случае — `skulking`).

```
assert(performAction("Fuma") === "Fuma skulking",
       "The default value is used for Fuma");

assert(performAction("Yoshi") === "Yoshi skulking",
       "The default value is used for Yoshi");

assert(performAction("Hattori") === "Hattori skulking",
       "The default value is used for Hattori");
```

А если значение параметра указывается при вызове функции, то оно заменяет его стандартное значение:

```
assert(performAction("Yagyu", "sneaking") === "Yagyu sneaking",
       "Yagyu can do whatever he pleases, even sneak!");
```

Параметрам функции можно присваивать любые стандартные значения: как простых, примитивных типов данных (например, числовые или строковые значения), так и сложных типов данных, в том числе объекты, массивы и даже функции. Эти значения обрабатываются при каждом вызове функции слева направо. Поэтому мы можем задать стандартное значение для параметра функции, использовав для него значение одного из предыдущих параметров, как демонстрируется в примере кода из листинга 3.10.

Листинг 3.10. Обращение к значению предыдущего параметра функции

```
function performAction(ninja, action = "skulking",
                       message = ninja + " " + action) {
  return message;
}

assert(performAction("Yoshi") === "Yoshi skulking",
       "Yoshi is skulking");
```

В качестве стандартных значений параметров можно указывать произвольные выражения и даже обращаться по ходу дела к значениям предыдущих параметров функции

Несмотря на то что в JavaScript допускается делать нечто подобное, следует, однако, иметь в виду, что такая норма практики программирования совсем не способствует повышению удобочитаемости исходного кода, и поэтому ее следует всячески избегать. Впрочем, умеренное употребление стандартных значений параметров как средства для исключения пустых значений или в качестве относительно просто устанавливаемых признаков, определяющих поведение функций, может сделать исходный код более простым и изящным.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Написание изощренного кода опирается на ясное представление о JavaScript как о языке функционального программирования.
- Функции являются объектами высшего порядка и трактуются как любые другие объекты в JavaScript. Подобно объектам любого другого типа, функции можно:
  - создавать с помощью литералов;
  - присваивать переменным или свойствам;
  - передавать в качестве параметров;
  - возвращать в качестве результатов выполнения других функций;
  - наделять свойствами и методами.
- Функции обратного вызова предназначены для последующего "обратного вызова" в коде и часто применяются, особенно при обработке событий.
- Свойствами функций можно выгодно пользоваться для хранения в них любой информации. Например, можно:
  - хранить одни функции в свойствах других функций для последующего обращения и вызова;
  - пользоваться свойствами функций для создания кеша значений, чтобы исключить излишние вычисления.
- Имеются разные типы определяемых функций: объявления функций, функциональные выражения, стрелочные функции и функции-генераторы.

- Объявления функций и функциональные выражения относятся к двум самым распространенным типам функций. Объявления функций должны непременно содержать имя функции и указываться в отдельных операторах программного кода. А функциональные выражения совсем не обязательно должны быть именованными, но должны быть частью другого оператора программного кода.
- Стрелочные функции являются новым языковым средством JavaScript, позволяющим определять функции в намного более краткой форме, чем стандартные функции.
- Параметр является переменной, указываемой в определении функции, тогда как аргумент – значением, передаваемым функции при ее вызове.
- Списки параметров и аргументов функции могут отличаться по длине:
  - неприсвоенные параметры считаются неопределенными (`undefined`);
  - лишние аргументы не привязываются к именам параметров.
- Оставшиеся параметры и стандартные значения параметров являются новыми языковыми средствами ES6:
  - оставшиеся параметры позволяют обращаться к остальным аргументам, у которых отсутствуют соответствующие имена параметров;
  - стандартные значения параметров позволяют задавать их значения, которые будут использоваться в том случае, если при вызове функции значение параметра не указано.

Упражнения

1. Какие функции в приведенном ниже фрагменте кода являются функциями обратного вызова?

```
numbers.sort(function sortAsc(a,b){
  return a - b;
});

function ninja(){}
ninja();

var myButton = document.getElementById("myButton");
myButton.addEventListener("click", function handleClick(){
  alert("Clicked");
});
```

2. Разделите функции в приведенном ниже фрагменте кода по типам, отнеся их к объявлению функции, функциональному выражению или стрелочной функции.

```
numbers.sort(function sortAsc(a,b){
  return a - b;
});
```

```
numbers.sort((a,b) => b - a);

(function(){})();

function outer(){
  function inner(){}
  return inner;
}

(function(){}());

(()=>"Yoshi")();
```

3. Какие значения примут переменные `samurai` и `ninja` после выполнения приведенного ниже фрагмента кода?

```
var samurai = (() => "Tomoe")();
var ninja = (() => {"Yoshi"})();
```

4. Какие значения принимают параметры `a`, `b` и `c` в теле функции `test()` в двух ее вызовах, приведенных ниже?

```
function test(a, b, ...c){ /*a, b, c*/}

test(1, 2, 3, 4, 5);
test();
```

5. Какие значения примут переменные `message1` и `message2` после выполнения приведенного ниже фрагмента кода?

```
function getNinjaWieldingWeapon(ninja, weapon = "katana"){
  return ninja + " " + weapon;
}

var message1 = getNinjaWieldingWeapon("Yoshi");
var message2 = getNinjaWieldingWeapon("Yoshi", "wakizashi");
```

4 Функции для ученика мастера: представление об их вызове

В этой главе...

- Два неявных параметра функции: `this` и `arguments`
- Способы вызова функций
- Разрешение затруднений, связанных с контекстами функций

В предыдущей главе было показано, что язык программирования JavaScript обладает значительными функционально-ориентированными характеристиками. В ней были исследованы отличия аргументов и параметров вызываемых функций, а также порядок передачи значений от аргументов к параметрам вызываемых функций.

Аналогичным образом в этой главе сначала обсуждаются следующие особенности функций, которых мы явно избегали в предыдущей главе: параметры `this` и `arguments`. Эти параметры негласно передаются функциям и могут быть доступны подобно любым другим явно именованным параметрам в теле функции.

Параметр `this` представляет контекст функции — объект, для которого эта функция вызывается, тогда как параметр `arguments` — все аргументы, передаваемые при вызове функции. Оба параметра играют очень важную роль в коде JavaScript. В частности, параметр `this` является одной из существенных составляющих объектно-ориентированного характера JavaScript, а параметр `arguments` позволяет творчески подходить к аргументам, передаваемым функциям. Именно по этой причине мы выясним типичные скрытые препятствия, связанные с обоими упоминаемыми здесь неявными параметрами.

После этого мы рассмотрим способы вызова функций в JavaScript. Ведь способ вызова функции оказывает существенное влияние на порядок определения ее неявных параметров. И в завершение этой главы мы исследуем характерные подводные камни, связанные с контекстом функции и параметром `this`. Итак, приступим к исследованию без лишних церемоний!

Знаете ли вы?

Почему параметр `this` называется *контекстом* функции?

Чем функция отличается от метода?

Что произойдет, если функция-конструктор явно возвратит объект?

4.1. Использование неявных параметров функции

В предыдущей главе мы исследовали отличия *параметров* функции (т.е. переменных, перечисляемых в определении функции) от ее *аргументов* (т.е. значений, передаваемых функции при ее вызове). Но мы не упомянули о том, что, помимо параметров, явно указываемых в определении функции, при ее вызове обычно передаются и два неявных параметра — `arguments` и `this`.

Под словом *неявный* здесь подразумевается, что такие параметры не перечисляются явно в сигнатуре функции, но по умолчанию передаются ей и находятся в ее области видимости. К ним можно обращаться в самой функции таким же образом, как и к любым другим явно именованным параметрам. Рассмотрим каждый из этих неявных параметров по очереди.

4.1.1. Параметр `arguments`

Этот параметр представляет собой коллекцию всех аргументов, передаваемых функции. Он удобен тем, что предоставляет доступ ко всем аргументам функции независимо от того, определены ли для них соответствующие параметры. Это дает возможность реализовать перегрузку функций — механизм, не поддерживаемый непосредственно в JavaScript, а также функции с переменным количеством аргументов. Но, откровенно говоря, благодаря введению в стандарте ES6 оставшихся параметров (`rest`), описанных в предыдущей главе, потребность в параметре `arguments` значительно сократилась. Тем не менее принцип действия параметра `arguments` очень важно знать, поскольку он нередко встречается в унаследованном коде.

У объекта `arguments` имеется свойство `length`, обозначающее точное количество аргументов. Значения отдельных аргументов могут быть получены путем индексирования массива. Так, по индексу `arguments[2]` из массива будет извлечен третий параметр функции. Рассмотрим в качестве примера код из листинга 4.1.

Листинг 4.1. Применение параметра `arguments`

```
function whatever(a, b, c){
```
Объявить функцию с тремя параметрами: a, b и c

```
  assert(a === 1, 'The value of a is 1');
  assert(b === 2, 'The value of b is 2');
  assert(c === 3, 'The value of c is 3');
```
Проверить правильность значений

```
  assert(arguments.length === 5,
         'We've passed in 5 parameters');
```
На самом деле функции передаются пять аргументов

```
  assert(arguments[0] === a,
         'The first argument is assigned to a');
  assert(arguments[1] === b,
         'The second argument is assigned to b');
  assert(arguments[2] === c,
         'The third argument is assigned to c');
```
Проверить, совпадают ли три первых аргумента с параметрами функции

```
  assert(arguments[3] === 4,
         'We can access the fourth argument');
  assert(arguments[4] === 5,
         'We can access the fifth argument');
}
```
Проверить, доступны ли "лишние" аргументы через параметр `arguments`

```
whatever(1,2,3,4,5);
```
Вызвать функцию с пятью аргументами

В данном примере определяется функция `whatever()`, вызываемая с пятью аргументами, `whatever(1, 2, 3, 4, 5)`, несмотря на то, что она объявлена лишь с тремя аргументами `a`, `b`, `c`, как показано ниже.

```
function whatever(a, b, c){
  ...
}
```

Три первых аргумента доступны через соответствующие им параметры функции `a`, `b`, `c` следующим образом:

```
assert(a === 1, 'The value of a is 1');
assert(b === 2, 'The value of b is 2');
assert(c === 3, 'The value of c is 3');
```

Чтобы проверить, сколько аргументов в общем было передано функции, можно также воспользоваться свойством `arguments.length`. Параметром `arguments` можно также воспользоваться для доступа к каждому аргументу в отдельности через индекс массива. Следует особо подчеркнуть, что к числу доступных относятся также оставшиеся аргументы, не связанные ни с одним из параметров функции, как выделено ниже полужирным.

```
assert(arguments[0] === a, 'The first argument is assigned to a');
assert(arguments[1] === b, 'The second argument is assigned to b');
assert(arguments[2] === c, 'The third argument is assigned to c');
```

```
assert(arguments[3] === 4, 'We can access the fourth argument');
assert(arguments[4] === 5, 'We can access the fifth argument');
```

Обращаться к параметру `arguments` все же не рекомендуется. Ведь вы можете неверно принять его за массив, поскольку у него имеется свойство `length`, а его элементы могут быть извлечены, как из массива. Тем не менее это не типичный для JavaScript массив, и если вы попытаетесь применить к параметру `arguments` методы обработки массивов (например, метод `sort()`, упоминавшийся в предыдущей главе), вас постигнет полное разочарование. Поэтому воспринимайте параметр `arguments` просто как *похожую на массив* конструкцию и старайтесь пользоваться ей как можно реже.

Как упоминалось выше, главное назначение объекта `arguments` — предоставить доступ ко всем аргументам, переданным функции независимо от того, связан ли отдельный аргумент с соответствующим параметром. В качестве примера в коде из листинга 4.2 показано, как реализовать функцию, вычисляющую сумму произвольного числа аргументов.

Листинг 4.2. Применение объекта `arguments` для выполнения операций над всеми аргументами функции

```
function sum() {                          ← Функция без явно определенных параметров
  var sum = 0;
  for(var i = 0; i < arguments.length; i++){   Перебрать все переданные аргументы
    sum += arguments[i];                       и получить доступ к отдельным элементам
  }                                            по индексу
  return sum;
}

assert(sum(1, 2) === 3, "We can add two numbers");
assert(sum(1, 2, 3) === 6, "We can add three numbers");          Вызвать функцию
assert(sum(1, 2, 3, 4) === 10, "We can add four numbers");       с произвольным
                                                                 числом аргументов
```

В данном случае сначала определяется функция `sum()` без единого явно определенного параметра. Тем не менее все ее аргументы доступны через объект `arguments`. В частности, можно перебрать все ее аргументы и вычислить их сумму.

Примечание

Как неоднократно упоминалось ранее, вместо параметра **`arguments`** можно выгодно воспользоваться параметром **`rest`**, который является реальным массивом. А это означает, что к нему можно применить всевозможные методы обработки массивов. И это дает определенное преимущество над параметром **`arguments`**. В качестве упражнения попробуйте переписать исходный код из листинга 4.2, воспользовавшись в нем параметром **`rest`** вместо параметра **`arguments`**.

Эта функция дает следующий выигрыш: мы можем вызвать ее с любым количеством аргументов. Поэтому мы можем проверить работоспособность данной функции для нескольких случаев. Истинный потенциал объекта `arguments`

проявляется в том, что он позволяет создавать более универсальные и удобные функции, с которыми легко обращаться в самых разных ситуациях.

Разобравшись в принципе действия объекта `arguments`, выясним те подводные камни, которые связаны с его применением.

Объект `arguments` в качестве псевдонима параметров функции

Параметр `arguments` обладает любопытной особенностью, он позволяет назначить параметрам функции псевдонимы. Так, если задать новое значение первого аргумента (`arguments[0]`), изменится и значение первого параметра. В качестве примера рассмотрим код из листинга 4.3.

Листинг 4.3. Объект `arguments` назначает параметрам функции псевдонимы

```
function infiltrate(person) {
  assert(person === 'gardener',
         'The person is a gardener');
assert(arguments[0] === 'gardener',
         'The first argument is a gardener');

arguments[0] = 'ninja';

assert(person === 'ninja',
       'The person is a ninja now');
assert(arguments[0] === 'ninja',
       'The first argument is a ninja');

person = 'gardener';

assert(person === 'gardener',
       'The person is a gardener once more');
assert(arguments[0] === 'gardener',
       'The first argument is a gardener again');
}

infiltrate("gardener");
```

Параметр `parameter` принимает значение `"gardener"`, передаваемое в качестве первого аргумента

Изменение объекта `arguments` приводит также к изменению соответствующего аргумента

Псевдоним действует в обоих направлениях

Как следует из данного примера кода, объект `arguments` назначает параметрам функции псевдонимы. В данном примере определяется функция `infiltrate()`, принимающая единственный параметр `person` и вызываемая с аргументом `gardener`. Значения аргумента `gardener` могут быть доступны через параметр `person` данной функции и объект `arguments`:

```
assert(person === 'gardener', 'The person is a gardener');
assert(arguments[0] === 'gardener', 'The first argument is a gardener');
```

В связи с тем что объект `arguments` назначает параметрам функции псевдонимы, изменение объекта `arguments` отражается также на соответствующем параметре функции следующим образом:

```
arguments[0] = 'ninja';

assert(person === 'ninja', 'The person is a ninja now');
assert(arguments[0] === 'ninja', 'The first argument is a ninja');
```

То же самое происходит и в обратном направлении. Так, изменив параметр, можно наблюдать изменение как в самом параметре, так и в объекте `arguments`:

```
person = 'gardener';

assert(person === 'gardener',
       'The person is a gardener once more');
assert(arguments[0] === 'gardener',
       'The first argument is a gardener again');
```

Исключение псевдонимов

Принцип назначения параметрам функции псевдонимов через объект `arguments` может вызвать недоразумения, и поэтому в JavaScript предоставляется возможность отказаться от псевдонимов, перейдя в *строгий режим*. И, как всегда, обратимся к простому примеру кода, приведенному в листинге 4.4.

Строгий режим

Этот режим был внедрен в стандарте ES5 языка JavaScript с целью изменить поведение интерпретаторов JavaScript таким образом, чтобы ошибки генерировались, а не обрабатывались негласно. В строгом режиме изменяется поведение ненадежных языковых средств, а действие некоторых из них вообще запрещается (подробнее об этом — далее в главе). К числу изменений, происходящих в строгом режиме, относится запрет на использование псевдонимов через объект **`arguments`**.

Листинга 4.4. Применение строгого режима с целью запретить назначение псевдонимов параметрам функции

```
"use strict";                          ← Активизировать строгий режим

function infiltrate(person){
  assert(person === 'gardener',
         'The person is a gardener');
  assert(arguments[0] === 'gardener',
         'The first argument is a gardener');

  arguments[0] = 'ninja';              ← Изменить первый аргумент

  assert(arguments[0] === 'ninja',
         'The first argument is now a ninja');

assert(person === 'gardener',
       'The person is still a gardener');
}

infiltrate("gardener");
```

Параметр `person` и первый аргумент имеют сначала одно и то же значение

Первый аргумент изменен

Значение параметра person не изменилось

В первой строке данного примера кода указана простая символьная строка "use strict", сообщающая интерпретатору JavaScript, что следующий далее код требуется выполнить в строгом режиме. В данном примере строгий режим изменяет семантику программы таким образом, чтобы параметр person и первый аргумент принимали сначала одинаковое значение, как показано ниже.

```
assert(person === 'gardener', 'The person is a gardener');
assert(arguments[0] === 'gardener',
       'The first argument is a gardener');
```

Но, в отличие от нестрогого режима, на этот раз объект arguments не назначает псевдонимы параметрам функции. Так, если изменить значение первого аргумента (arguments[0] = 'ninja'), то изменится только первый аргумент, но не параметр person:

```
assert(arguments[0] === 'ninja',
       'The first argument is now a ninja');
assert(person === 'gardener',
       'The person is still a gardener');
```

Мы еще вернемся к объекту arguments далее в данной книге, а пока что рассмотрим еще один неявный параметр this. В какой-то степени этот параметр представляет еще больший интерес.

4.1.2. Параметр this, представляющий контекст функции

Всякий раз, когда вызывается функция, помимо параметров, представляющих аргументы, явно указанные при вызове функции, ей также передается неявный параметр this. Параметр this служит важной составляющей объектно-ориентированного характера JavaScript, обозначая объект, связанный с вызовом функции. Именно по этому он нередко называется *контекстом функции*.

Те, кто перешел на JavaScript из таких языков объектно-ориентированного программирования, как Java, могут превратно трактовать понятие контекста функции, считая, что параметр this указывает на экземпляр объекта того класса, в котором определен его метод. Но, как будет показано далее, вызов функции как *метода* в JavaScript — это лишь один из способов ее вызова. На самом деле то, на что указывает параметр this, определяется, в отличие от Java или C#, не тем, как функция объявляется, а тем, как она *вызывается*.

Ясное представление об истинном характере параметра this служит одним из важных опорных моментов для понимания объектно-ориентированного характера JavaScript. Поэтому в данной книге будут рассмотрены различные способы вызова функции и, в частности, основные их отличия, а также порядок определения значения параметра this. После этого мы снова вернемся к контекстам функций, чтобы более основательно исследовать их. Поэтому не особенно тревожьтесь, если ясное представление о контексте функции не сложилось у вас окончательно. Мы еще обсудим его более подробно, а до тех пор поясним, как вызываются функции.

4.2. Вызов функций

Вам, вероятно, не раз приходилось вызывать функции в коде JavaScript, но задумывались ли вы над тем, что в действительности при этом происходит? Оказывается, что способ вызова функции существенно влияет на порядок выполнения кода в ней и особенно на установку параметра `this`. Это отличие имеет гораздо большее значение, чем кажется на первый взгляд. Поэтому мы подробно рассмотрим его в текущем разделе, чтобы выгодно воспользоваться им в остальной части книги и тем самым помочь вам повысить свои навыки программирования на JavaScript до уровня настоящего мастера.

Итак, имеются четыре способа вызова функции, каждому из которых присущи свои особенности.

- **Как функция.** Например, `skulk()` — это функция, вызываемая непосредственно.
- **Как метод.** Например, вызов `ninja.skulk()` тесно связан с объектом, допуская объектно-ориентированное программирование.
- **Как конструктор.** Например, в операции `new Ninja()` создается новый объект.
- **Через методы `apply()` и `call()` отдельной функции.** Например, `skulk.call(ninja)` или `skulk.apply(ninja)`.

Ниже приведены наглядные примеры применения перечисленных выше способов вызова функций.

```
function skulk(name) {}
function Ninja(name) {}

skulk('Hattori');
(function(who){ return who; })('Hattori');

var ninja = {
  skulk: function(){}
};

ninja.skulk('Hattori');

ninja = new Ninja('Hattori');

skulk.call(ninja, 'Hattori');

skulk.apply(ninja, ['Hattori']);
```

Вызов как функции

Вызов как метода `ninja()`

Вызов как конструктора

Вызов через метод `call()`

Вызовчерез метод `apply()`

В о всех перечисленных выше способах, кроме двух последних, операция вызова функции обозначена парой круглых скобок, следующих после выражения, в котором определяется ссылка на функцию. Итак, начнем исследование способов вызова функций с простейшей формы: вызова как функции.

4.2.1. Вызов как функции

Что же означает вызов как функции? Разумеется, функции вызываются как *функции*, и было бы неразумно думать иначе. Но в действительности обозначение "вызов как функции" выбрано для того, чтобы каким-то образом провести различие между данным способом и другими механизмами вызова функций. Таким образом, если функция не вызывается как метод, конструктор или с помощью метода `apply()` или `call()`, это просто означает, что она вызывается как функция.

Подобного рода вызов происходит и в том случае, когда функция вызывается с помощью операции `()`, а выражение, к которому эта операция применяется, не ссылается на функцию как на свойство объекта. (В последнем случае функция вызывалась бы как метод, но об этом способе вызова функций речь пойдет далее.) Ниже приведены простые примеры вызовов как функции.

```
function ninja(){};
ninja();
```

Объявление функции и ее вызов как функции

Функциональное выражение, немедленно вызываемое в виде функции

```
var samurai = function(){};
samurai();
(function(){})()
```

Функциональное выражение, вызываемое как функция

Когда функция вызывается таким способом, ее контекст (т.е. значение параметра `this`) может быть двояким. В нестрогом режиме контекст может быть глобальным (объект `window`), а в строгом режиме — неопределенным (`undefined`).

В примере кода из листинга 4.5 наглядно демонстрируются отличия строгого режима от нестрогого.

Листинг 4.5. Вызов как функции

```
function ninja() {
  return this;
}
```

Функция в нестрогом режиме

```
function samurai() {
  "use strict";
  return this;
}
```

Функция в строгом режиме

```
assert(ninja() === window,
       "In a 'nonstrict' ninja function, " +
       "the context is the global window object");
```

Как и предполагалось, контекстом функции в нестрогом режиме является объект `window`

```
assert(samurai() === undefined,
       "In a 'strict' samurai function, " +
       "the context is undefined");
```

А в строгом режиме контекст функции не определен

Примечание

Как видите, строгий режим, как правило, оказывается намного более простым, чем нестрогий. Так, если функция вызывается в листинге 4.5 как функция, а не метод, вместе с ней не указывается объект, для которого она должна вызываться. Поэтому, на наш взгляд, целесообразнее устанавливать неопределенное (`undefined`) значение параметра `this`, как в строгом режиме, а не глобальный объект `window`, как в нестрогом режиме. В общем, многие из подобных мелких несуразностей устраняются в строгом режиме работы JavaScript. Достаточно вспомнить назначение псевдонимов параметрам функции, упоминавшееся в начале этой главы.

Вам, вероятно, не раз приходилось писать подобный код, даже не задумываясь об этом. А теперь пойдем дальше, чтобы выяснить, каким образом функции вызываются *как методы*.

4.2.2. Вызов как метода

Когда функция присваивается свойству объекта *и* вызов происходит по ссылке на функцию с помощью этого свойства, функция вызывается как *метод* данного объекта. Ниже приведен пример такого вызова.

```
var ninja = {};
ninja.skulk = function(){};
ninja.skulk();
```

Что же из этого следует? Функция в данном случае называется "методом", но что в этом любопытного или полезного? Если у вас имеется некоторый опыт объектно-ориентированного программирования, то вы, вероятно, вспомните, что объект, к которому относится вызываемый метод, доступен в теле метода по ссылке `this`. То же самое происходит и здесь. Когда функция вызывается как *метод* объекта, этот объект становится контекстом функции и доступен в ней через параметр `this`. И это одно из основных средств для написания объектно-ориентированного кода на JavaScript. (Другим средством является рассматриваемый далее конструктор.)

В качестве примера рассмотрим код из листинга 4.6, чтобы продемонстрировать отличия и сходства в вызовах функций как функции и как метода.

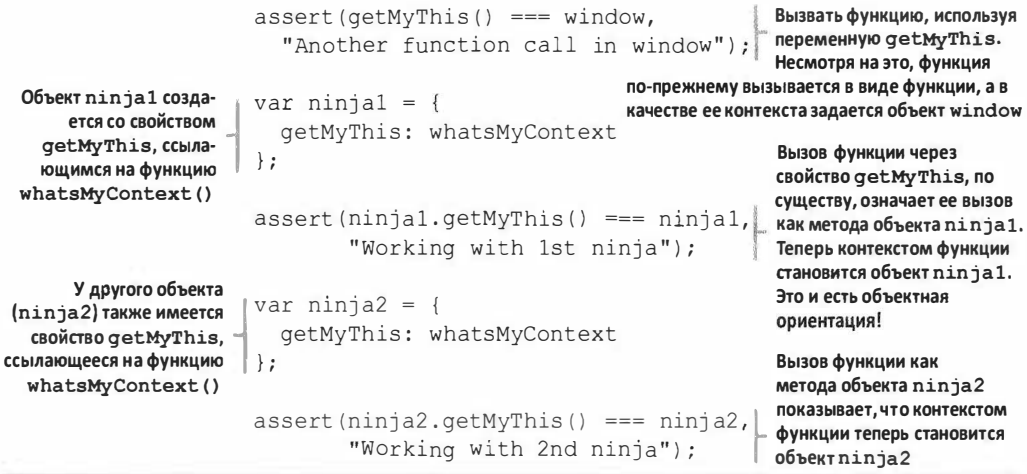
Листинг 4.6. Отличия и сходства в вызовах функций как функций и как методов

```
function whatsMyContext() {
  return this;
}

assert(whatsMyContext() === window,
       "Function call on window");

var getMyThis = whatsMyContext;
```

Возвращается контекст функции, позволяющий исследовать его извне

В результате вызова функции в качестве ее контекста задается объект `window`

Свойству `getMyThis` присваивается ссылка на функцию `whatsMyContext()`



В данном примере задается функция `whatsMyContext()`, применяемая для тестирования остальной части кода из листинга 4.6. Эта функция лишь возвращает свой контекст, как показано ниже, чтобы во внешней функции можно было видеть, в каком именно контексте происходит вызов данной функции. В противном случае узнать контекст функции было бы затруднительно.

```
function whatsMyContext() {
  return this;
}
```

Когда функция вызывается непосредственно по имени, она вызывается как функция. И тогда можно предположить, что контекстом функции станет глобальный объект `window`, поскольку это происходит в нестрогом режиме. В этом можно убедиться следующим образом:

```
assert(whatsMyContext() === window, ...)
```

Затем переменной `getMyThis` присваивается ссылка на функцию `whatsMyContext()`:

```
var getMyThis = whatsMyContext;
```

При этом создается не второй экземпляр данной функции, а только ссылка на нее. Ведь функции, как известно, являются объектами высшего порядка.

Когда функция вызывается через переменную, а это вполне допустимо, поскольку операцию вызова функции можно выполнять над любым выражением, которое интерпретируется как функция, то она опять же вызывается как функция. А раз так, то в качестве контекста функции предполагается объект `window`, как показано ниже.

```
assert(getMyThis() === window,
       "Another function call in window");
```

А далее дело немного усложняется, когда переменной `ninja1` присваивается объект со свойством `getMyThis`, получающим ссылку на функцию

whatsMyContext(), как показано ниже. Это, по существу, означает, что для объекта ninja1 был создан *метод* getMyThis(), а не то, что функция whatsMyContext() *стала* методом этого объекта. Как было показано ранее, функция whatsMyContext() является независимой и может вызываться самыми разными способами.

```
var ninja1 = {
  getMyThis: whatsMyContext
};
```

Как было сказано ранее, при вызове функции через метод объекта в качестве контекста функции предполагается объект этого метода (в данном случае объект ninja1). Именно это и утверждается ниже.

```
assert(ninja1.getMyThis() === ninja1,
       "Working with 1st ninja");
```

Примечание

Вызов функций в качестве методов играет решающую роль в объектно-ориентированном способе программирования на JavaScript. Ведь это дает возможность пользоваться ссылкой **this** в любом методе для обращения к объекту, "владеющему" этим методом, и является основополагающим принципом объектно-ориентированного программирования.

Чтобы довести дело до логического завершения, тестирование в данном примере продолжается созданием еще одного объекта ninja2 также со свойством getMyThis, получающим ссылку на функцию whatsMyContext(). После вызова этой функции через объект ninja2 совершенно верно утверждается, что ее контекстом становится объект ninja2:

```
var ninja2 = {
  getMyThis: whatsMyContext
};

assert(ninja2.getMyThis() === ninja2,
       "Working with 2nd ninja");
```

И хотя это та же самая функция whatsMyContext(), которая используется повсюду в данном примере, ее контекст, возвращаемый через параметр this, изменяется в зависимости от способа ее вызова. Например, одна и та же функция оказывается общей для обоих объектов, ninja1 и ninja2, хотя когда она выполняется, она получает доступ и может производить операции над тем объектом, метод которого вызывается. Для выполнения одной и той же обработки по разным объектам совсем не обязательно создавать отдельные копии функции. И в этом также заключается один из основополагающих принципов объектно-ориентированного программирования.

Безусловно, это довольно эффективная возможность, но ее использование в данном примере имеет свои ограничения. Прежде всего, создав два объекта, ninja1 и ninja2, мы получили возможность воспользоваться одной и той же

функцией в качестве метода каждого из них. Но при этом нам пришлось немного повторить код, чтобы создать отдельные объекты и их методы.

Впрочем, это обстоятельство не должно вас обескураживать, ведь в JavaScript предоставляются механизмы, позволяющие создавать объекты по одному шаблону намного проще, чем в рассмотренном здесь примере. Все эти механизмы будут обсуждаться более подробно в главе 7, а до тех пор рассмотрим еще один способ вызова функций *как конструктора*.

4.2.3. Вызов как конструктора

В функции, которую предполагается использовать в качестве конструктора, нет ничего особенного. Функции-конструкторы объявляются таким же образом, как и любые другие функции, и мы можем легко воспользоваться объявлениями функций и функциональными выражениями для конструирования новых объектов. Единственное исключение из этого правила составляет стрелочная функция, которая действует несколько иначе, как поясняется далее в этой главе. Но в любом случае главное отличие состоит в способе вызова функции.

Для вызова функции как *конструктора* перед ее именем указывается ключевое слово `new`. Для примера обратимся к функции `whatsMyContext()`, упоминавшейся в предыдущем разделе.

```
function whatsMyContext(){ return this; }
```

Если функцию `whatsMyContext()` требуется вызвать как конструктор, для этого достаточно написать следующую строку кода:

```
new whatsMyContext();
```

Но даже если вызвать функцию `whatsMyContext()` как конструктор, она окажется не совсем пригодной для применения в качестве конструктора. Попробуем выяснить причины этого, обсудив особенности конструкторов.

Примечание

При обсуждении способов определения функций в главе 3, кроме объявлений функций, функциональных выражений, стрелочных функций и функций-генераторов, упоминались также *конструкторы функций*, позволяющие создавать новые функции из символьных строк. Например, в следующей строке кода создается новая функция, которой передаются два параметра, `a` и `b`, а она возвращает их сумму:

```
new Function('a', 'b', 'return a + b')
```

Такие *конструкторы функций* не следует путать с *функциями-конструкторами*! Отличия одних от других едва заметны, но они весьма существенны. С одной стороны, конструкторы функций позволяют создавать функции динамически из формируемых символьных строк. А с другой стороны, функции-конструкторы служат для создания и инициализации экземпляров объектов.

Сильные стороны конструкторов

Вызов функции как конструктора считается весьма эффективным языковым средством JavaScript, как демонстрируется в примере кода из листинга 4.7.

Листинг 4.7. Создание новых объектов с помощью конструктора

```
function Ninja() {
  this.skulk = function() {
    return this;
  };
}

var ninja1 = new Ninja();
var ninja2 = new Ninja();

assert(ninja1.skulk() === ninja1,
       "The 1st ninja is skulking");
assert(ninja2.skulk() === ninja2,
       "The 2nd ninja is skulking");
```

Создать два объекта, вызвав конструктор с помощью операции new. Вновь созданные объекты доступны по ссылкам в переменных ninja1 и ninja2

Конструктор, создающий свойство skulk того объекта, который является контекстом функции. Метод skulk() снова возвращает контекст функции, чтобы проверить ее извне

Проверить метод skulk() созданных объектов. В каждом случае он должен возвратить свой созданный объект

В данном примере кода создается функция `Ninja()`, предназначенная для конструирования объектов скрывающихся ниндзя. При ее вызове с ключевым словом `new` создается экземпляр пустого объекта, который передается функции в качестве параметра `this`. Конструктор создает для данного объекта свойство `skulk`, которому присваивается функция. В итоге эта функция превращается в метод вновь созданного объекта.

В общем, при вызове конструктора производятся особые действия, как показано на рис. 4.1. А при вызове функции с ключевым словом `new` инициируются следующие действия.

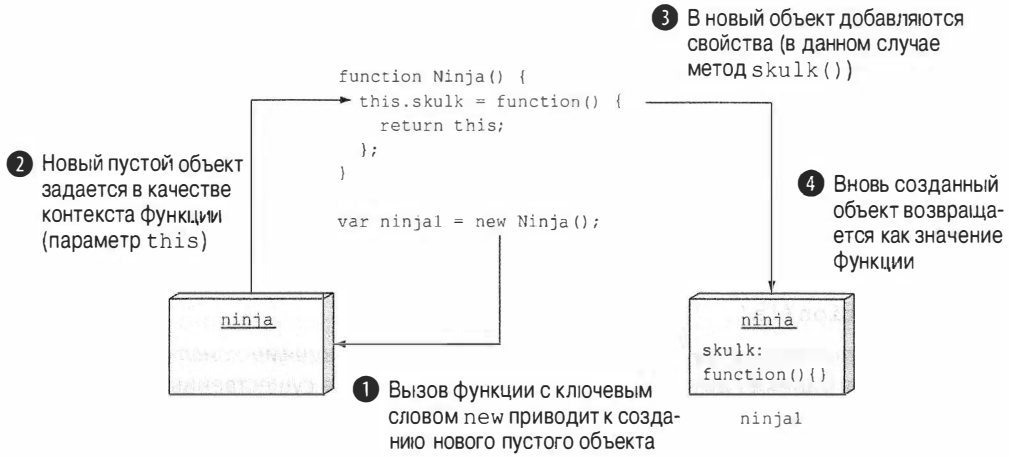


Рис. 4.1. При вызове функции с ключевым словом **new** создается новый пустой объект, который задается в качестве контекста функции-конструктора в параметре **this**

1. Создается новый пустой объект.
2. Этот объект передается конструктору в качестве параметра `this`, а следовательно, он становится контекстом функции данного конструктора.
3. Вновь сконструированный объект возвращается в качестве значения операции `new`, за одним рассматриваемым ниже исключением.

Два последних действия объясняют, почему функцию `whatsMyContext()` не стоит вызывать как конструктор в операции `new whatsMyContext()`. Назначение конструктора — создать новый объект, проинициализировать и возвратить его в качестве значения конструктора. Все, что мешает достижению этой цели, не годится для функций, предназначенных для применения в качестве конструкторов.

Рассмотрим следующий пример более подходящей для этой цели функции-конструктора объектов типа `Ninja`, где устанавливаются объекты скрывающихся ниндзя:

```
function Ninja() {
  this.skulk = function() {
    return this;
  };
}
```

Метод `skulk()` выполняет ту же самую операцию, что и функция `whatsMyContext()` из предыдущего раздела, возвращая контекст функции, который может быть проверен извне. Определив конструктор, можно создать два новых объекта, дважды вызвав конструктор объектов типа `Ninja`. Как показано ниже, значения, возвращаемые в результате этих вызовов, сохраняются в переменных, которые содержат ссылки на вновь созданные объекты типа `Ninja`.

```
var ninja1 = new Ninja();
var ninja2 = new Ninja();
```

Затем выполняются приведенные ниже тесты с целью проверить, происходит ли должным образом воздействие на объект при каждом вызове метода.

```
assert(ninja1.skulk() === ninja1,
       "The 1st ninja is skulking");
assert(ninja2.skulk() === ninja2,
       "The 2nd ninja is skulking");
```

Вот, собственно, и все! Теперь вы знаете, как создавать и инициализировать новые объекты с помощью функций-конструкторов. В результате вызова функции с ключевым словом `new` возвращается вновь созданный объект. Но давайте проверим, всегда ли это именно так.

Значения, возвращаемые конструкторами

Как упоминалось ранее, конструкторы предназначены для инициализации вновь созданных объектов, которые возвращаются в результате вызова конструктора (через операцию `new`). Но что произойдет, если конструктор воз-

вратит некоторое значение? Исследуем эту ситуацию на примере кода из листинга 4.8.

Листинг 4.8. Конструкторы, возвращающие значения примитивных типов

```
function Ninja() {
  this.skulk = function () {
    return true;
  };

  return 1;
}

assert(Ninja() === 1,
  "Return value honored when not called as a constructor");

var ninja = new Ninja();

assert(typeof ninja === "object",
       "Object returned when called as a constructor");
assert(typeof ninja.skulk === "function",
       "ninja object has a skulk method");
```

Определить функцию-конструктор объектов типа `Ninja`

Конструктор возвращает конкретное значение примитивного типа — число 1

Функция вызывается в виде функции и возвращает, как и предполагалось, значение 1

Функция вызывается как конструктор через операцию `new`

Убедиться, что возвращаемое значение 1 игнорируется, а из операции `new` возвращается новый инициализированный объект

Если выполнить код из листинга 4.8, то он окажется вполне работоспособным. Тот факт, что функция-конструктор объектов типа `Ninja` возвращает простое числовое значение **1**, не оказывает существенного влияния на поведение кода. Если вызвать ее как функцию, она возвратит, как и предполагалось, значение **1**. А если вызвать ее как конструктор, т.е. с ключевым словом `new`, то будет создан и возвращен новый объект `Ninja`. Итак, все пока что идет нормально.

А теперь попробуем сделать нечто иное, создав функцию-конструктор, возвращающую другой объект, как демонстрируется в примере кода из листинга 4.9.

Листинг 4.9. Конструкторы, возвращающие значения объектов

```
var puppet = {
  rules: false
};

function Emperor() {
  this.rules = true;
  return puppet;
}

var emperor = new Emperor();

assert(emperor === puppet,
       "The emperor is merely a puppet!");
assert(emperor.rules === false,
       "The puppet does not know how to rule!");
```

Создать глобальный объект, свойство `rules` которого равно `false`

Возвратить этот объект, несмотря на инициализацию объекта, переданного по ссылке `this`

Вызвать функцию как конструктор

Убедиться, что переменной `emperor` присваивается объект, возвращаемый конструктором, а не объект, созданный в операции `new`

В данном примере предпринят несколько иной подход. Сначала в коде из данного примера создается глобальный объект, ссылка на который сохраняется в переменной `puppet`, со свойством `rules`, принимающим логическое значение `false`:

```
var puppet = {
  rules: false
};
```

Затем создается функция-конструктор объектов типа `Emperor`, добавляющая ко вновь сконструированному объекту свойство `rules`, принимающее значение `true`. Кроме того, функции-конструктору объектов типа `Emperor` присуща следующая особенность возвращать глобальный объект `puppet`:

```
function Emperor() {
  this.rules = true;
  return puppet;
}
```

Далее эта функция вызывается как конструктор с ключевым словом `new`, как показано ниже.

```
var emperor = new Emperor();
```

Таким образом, возникает неоднозначная ситуация: сначала получается один объект, который передается конструктору в качестве контекста функции через параметр `this` и соответственно инициализируется, а затем возвращается совершенно другой объект `puppet`. Какой же объект правит бал? Ответ на этот вопрос дают следующие проверки:

```
assert(emperor === puppet,
       "The emperor is merely a puppet!");
assert(emperor.rules === false,
       "The puppet does not know how to rule!");
```

Как следует из этих проверок, объект `puppet` возвращается в качестве значения вызываемого конструктора, а вся инициализация, выполненная в конструкторе по контексту функции, оказалась напрасной. Объект `puppet` разоблачен!

Итак, произведя ряд проверок, можно сделать следующие выводы.

- Если конструктор возвращает объект, этот объект возвращается в виде значения всей операции `new`, а вновь созданный объект, передаваемый конструктору в качестве параметра `this`, игнорируется.
- Но если конструктор возвращает не объект, то возвращаемое значение игнорируется, а вместо него возвращается вновь созданный объект.

В силу этих особенностей функции, предназначенные служить в качестве конструкторов, как правило, кодируются иначе, чем остальные функции. Поэтому рассмотрим этот вопрос более подробно.

Соображения по поводу кодирования функций-конструкторов

Как пояснялось выше, назначение конструктора — инициализировать исходными данными новый объект, который будет создан при вызове функции. А поскольку такие функции *можно* вызывать как обычные функции и даже присваивать их свойствам объектов для вызова как методы, то они, как правило, не совсем подходят для применения в качестве конструкторов. Рассмотрим следующий пример функции:

```
function Ninja() {
  this.skulk = function() {
    return this;
  };
}
var whatever = Ninja();
```

Функцию-конструктор объектов типа `Ninja` можно вызвать как обычно, но тогда свойство `skulk` будет создано для объекта `window` в нестрогом режиме, и пользы от такой операции не особенно много. Дело обстоит еще более скверно в строгом режиме, где параметр `this` оказывается неопределенным и прикладной код JavaScript завершается аварийно. Но это даже хорошо, ведь если мы совершим подобную оплошность в нестрогом режиме, то можем и не заметить ее в отсутствие тщательных тестов, тогда как в строгом режиме такая оплошность не останется незамеченной. И этот пример наглядно показывает, почему был внедрен строгий режим.

Функции-конструкторы обычно кодируются и используются совсем иначе, чем другие функции, и пользы от них зачастую оказывается немного, если только они не вызываются как конструкторы. Поэтому были выработаны определенные соглашения по присвоению имен, чтобы как-то отличать функции-конструкторы от обычных функций и методов. И вы, возможно, уже обратили на это внимание.

Для имен функций обычно выбираются английские глаголы, описывающие их назначение, и сами имена начинаются со строчной буквы (например, `skulk()`, `creep()`, `sneak()`, `doSomethingWonderful()` и т.д.). Для имен конструкторов обычно выбираются английские существительные, описывающие конструируемый объект, и сами имена начинаются с прописной буквы (например, `Ninja()`, `Samurai()`, `Ronin()`, `Emperor()` и т.д.)

Нетрудно заметить, что конструктор намного упрощает создание многих объектов по одному и тому же шаблону, без многократного повторения одного и того же кода. В этом случае общий код пишется лишь один раз в теле самого конструктора. Более подробно о применении конструкторов и других механизмов объектно-ориентированного программирования, поддерживаемых в JavaScript, речь пойдет в главе 7. Эти языковые средства значительно упрощают создание шаблонов для объектов.

Но мы еще не завершили рассмотрение способов вызова функций. Ведь в JavaScript имеется еще один способ, предоставляющий немало возможностей

для тщательного контроля над вызовом функций. И этот способ будет рассмотрен ниже.

4.2.4. Вызов через методы `apply()` и `call()`

Как было показано ранее, главное отличие в способах вызова функций заключается в том, какой именно объект становится контекстом функции, на который ссылается параметр `this`, неявно передаваемый выполняющейся функции. Для методов это объект, который ими владеет, для функций верхнего уровня — объект `window` или неопределенное значение (`undefined`) в зависимости от текущей строгости режима, а для конструкторов — экземпляр вновь созданного объекта.

Но что, если требуется задать такой объект явным образом? Чтобы выяснить, зачем вообще нужна такая возможность, рассмотрим практический пример, иллюстрирующий, как ни странно, типичную программную ошибку, связанную с обработкой событий. Более подробно вопросы обработки событий обсуждаются в главе 13, а до тех пор будем считать, что при вызове обработчика событий в качестве контекста функции задается объект, к которому привязано наступившее событие. Итак, рассмотрим пример кода из листинга 4.10.

Листинг 4.10. Привязка конкретного контекста к функции

```
<button id="test">Click Me!</button>
<script>
function Button(){
  this.clicked = false;
  this.click = function(){
    this.clicked = true;
    assert(button.clicked, "The button has been clicked");
  };
}
var button = new Button();
var elem = document.getElementById("test");
elem.addEventListener("click", button.click);
</script>
```

Функция-конструктор, создающая объекты, сохраняющие состояние, связанное с кнопкой. С ее помощью можно отслеживать нажатие кнопки

Элемент разметки кнопки, которому присваивается обработчик событий

Объявить метод, предназначенный для обработки событий от щелчков на экранной кнопке. А поскольку это метод объекта, то мы можем использовать в теле функции переменную `this` для получения ссылки на объект

В теле метода проверяется, насколько верно изменилось состояние экранной кнопки после щелчка на ней

Создать экземпляр для отслеживания факта нажатия кнопки

Установить обработчик событий от щелчков на экранной кнопке

В данном примере с помощью HTML-кода (`<button id="test">Click Me!</button>`) создается экранная кнопка и требуется отслеживать моменты, когда на ней производится щелчок. Для сохранения информации об этом состоянии служит приведенная ниже функция-конструктор, создающая вспомогательный объект `button`, где предполагается хранить состояние нажатия экранной кнопки.

```
function Button(){
  this.clicked = false;
  this.click = function(){
    this.clicked = true;
```

```
    assert(button.clicked, "The button has been clicked");
  };
}
var button = new Button();
```

В этом объекте определяется также метод `click()`, предназначенный в качестве обработчика событий и вызываемый при нажатии экранной кнопки. В нем сначала присваивается логическое значение `true` свойству `clicked`, а затем проверяется, было ли состояние кнопки надлежащим образом зарегистрировано во вспомогательном объекте. Ради удобства здесь намерено используется идентификатор `button` вместо ключевого слова `this`, ведь они в любом случае должны обозначать одно и то же. И, наконец, метод `button.click()` устанавливается в качестве обработчика событий от щелчков на экранной кнопке, как показано ниже.

```
var elem = document.getElementById("test");
elem.addEventListener("click", button.click);
```

Если загрузить код из данного примера в браузер и щелкнуть на экранной кнопке, на экране появится не совсем верный результат, приведенный на рис. 4.2. В частности, перечеркнутый текст обозначает, что тест не прошел. Код из листинга 4.10 оказывается неработоспособным потому, что контекст функции `click()` *не* ссылается на объект `button`, как, собственно, и предполагалось.

Рис. 4.2. Почему же тест не прошел и чье состояние было изменено? Как правило, контекстом для функции обратного вызова служит объект, инициирующий событие (в данном случае это размеченная в коде HTML экранная кнопка, а не объект **`button`**)

Извлекая уроки из предыдущей главы, следует заметить, что если обратиться к функции через вызов `button.click()`, то ее контекстом станет экранная кнопка, поскольку данная функция вызывается как метод для объекта `button`. Но в данном примере система обработки событий в браузере определяет в качестве контекста вызова целевой элемент разметки события, и в конечном

итоге этим контекстом становится элемент разметки <button>, а не объект button. Таким образом, состояние click устанавливается не для того объекта!

Как ни странно, такое затруднение весьма типично, и далее в этой главе будут представлены способы полностью избежать его. А до тех пор выясним, как разрешить данное затруднение, установив явным образом контекст функции с помощью методов apply() и call().

Применение методов apply() и call()

В языке JavaScript предоставляются средства для вызова функции и явного указания любого объекта, который должен служить в качестве контекста функции. Это делается с помощью одного из двух методов, существующих для каждой функции: apply() и call(). И мы не оговорились: именно методов функций. Ведь функции — это объекты высшего порядка, создаваемые, между прочим, с помощью конструктора объектов типа Function. Следовательно, у них могут быть свойства и методы, как и у объектов любого другого типа.

Для вызова функции с помощью метода apply() последнему передаются два параметра: объект, назначаемый в качестве контекста функции, а также массив значений, используемых в качестве аргументов вызываемой функции. Аналогичным образом используется и метод call(), за исключением того, что аргументы передаются функции в списке, а не в массиве. В листинге 4.11 демонстрируется, как эти методы действуют непосредственно в коде.

Листинг 4.11. Назначение контекста функции с помощью методов apply() и call()

Функция juggle() манипулирует аргументами, сохраняя результат в том объекте, который становится контекстом функции

```
function juggle() {
  var result = 0;
  for (var n = 0; n < arguments.length; n++) {
    result += arguments[n];
  }
  this.result = result;
}

var ninja1 = {};
var ninja2 = {};

juggle.apply(ninja1,[1,2,3,4]);
juggle.call(ninja2, 5,6,7,8);

assert(ninja1.result === 10, "juggled via apply");
assert(ninja2.result === 26, "juggled via call");
```

Эти объекты первоначально пусты, а далее они служат в качестве субъектов тестирования

Вызвать метод call(), передав ему объект ninja2 и список аргументов

Вызвать метод apply(), передав ему объект ninja1 и массив аргументов

Как показывают тесты, результат манипулирования аргументами размещается в объектах, передаваемых методам

В данном примере сначала создается функция juggle(), в которой манипулирование определяется как суммирование всех аргументов и последующее их сохранение в свойстве result объекта, служащего в качестве контекста данной функции, обозначаемого ключевым словом this. Возможно, это и не совсем убедительное определение манипулирования, тем не менее оно позволит выяс-

нить, были ли аргументы переданы функции правильно и какой именно объект стал ее контекстом.

Далее создаются два объекта (`ninja1` и `ninja2`), которые станут контекстами функции. Первый из них передается методу `apply()` данной функции вместе с массивом аргументов, а второй — методу `call()` вместе со списком аргументов, как показано ниже.

```
juggle.apply(ninja1,[1,2,3,4]);
juggle.call(ninja2, 5,6,7,8);
```

Как видите, методы `apply()` и `call()` отличаются лишь способом передачи им аргументов. В частности, после контекста функции методу `apply()` передается массив аргументов, тогда как методу `call()` — список аргументов (рис. 4.3).

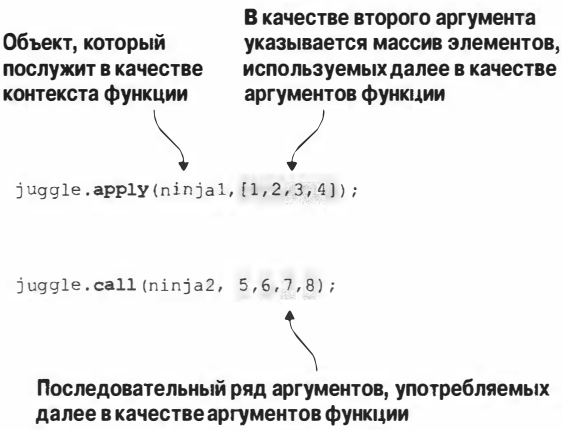


Рис. 4.3. В качестве первого аргумента методам **`apply()`** и **`call()`** передается объект, служащий в качестве контекста функции. Их вызовы отличаются лишь последующими аргументами. Так, методу **`apply()`** передается только один дополнительный аргумент (в виде массива значений), а методу **`call()`** — целый ряд аргументов, используемый далее в качестве аргументов функции

После передачи функции ее контекста и списка аргументов начинается тестирование, как показано ниже. Сначала проверяется, получает ли объект `ninja1`, обращение к которому происходит через метод `apply()`, свойство `result`, где сохраняется результат сложения значений (**1**, **2**, **3**, **4**) всех аргументов из переданного массива. Затем то же самое проделывается с объектом `ninja2`, обращение к которому происходит через метод `call()`, где проверяется результат сложения значений аргументов **5**, **6**, **7** и **8**. На рис. 4.4 более подробно показано, что же происходит в коде из листинга 4.11.

```
assert(ninja1.result === 10, "juggled via apply");
assert(ninja2.result === 26, "juggled via call");
```

```
function juggle() {
  var result = 0;
  for (var n = 0; n < arguments.length; n++){
    result += arguments[n];
  }
  this.result = result;
}

var ninja1 = {};
var ninja2 = {};
```

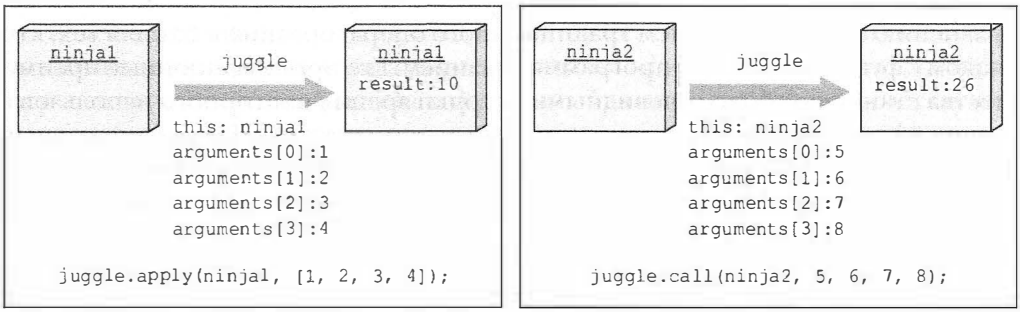


Рис. 4.4. Установка вручную контекста функции из листинга 4.11 с помощью встроенных методов **`call()`** и **`apply()`** приводит к показанному на этом рисунке сочетанию значений параметров **`this`** (контекстов функции) и **`arguments`**

Методы `call()` и `apply()` очень удобны в тех случаях, когда обычный контекст функции целесообразно приспособить под свои нужды, выбрав для него собственный объект. И особенно удобным это может оказаться при обращении к функциям обратного вызова.

Принудительная установка контекста в функциях обратного вызова

Рассмотрим конкретный пример принудительной установки специально избранного объекта в качестве контекста функции. С этой целью создадим простую функцию, которая будет выполнять определенную операцию над каждым элементом массива. В императивном программировании массив зачастую передается методу, а для обращения к элементам массива организуется цикл `for`, в котором над каждым элементом выполняется требующаяся операция, как показано ниже.

```
function(collection) {
  for (var n = 0; n < collection.length; n++) {
    /* сделать что-нибудь с элементом массива collection[n] */
  }
}
```

С другой стороны, в функциональном программировании обычно создается функция для выполнения операции над одним элементом массива. И этой функции в качестве аргумента передается по очереди каждый элемент массива:

```
function(item){
  /* сделать что-нибудь с заданным элементом */
}
```

Отличие заключается в том уровне, на котором функции мыслятся как стандартные блоки программы, а не императивные операторы. На первый взгляд это кажется слишком далеким от практики, поскольку мы лишь переносим цикл `for` на другой уровень.

Для упрощения стиля функционального программирования всем объектам массивов доступен метод `forEach()`, делающий обратный вызов для каждого элемента массива. Зачастую это более краткий и предпочтительный стиль по сравнению с использованием традиционного оператора цикла `for` для тех, кто знаком с функциональным программированием. Его организационные преимущества станут еще более очевидными (с точки зрения повторного использования кода), как только мы рассмотрим замыкания в главе 5. Такая итеративная функция могла бы передавать текущий элемент массива в качестве параметра функции обратного вызова, но чаще всего текущий элемент массива делается контекстом функции обратного вызова.

Несмотря на то что во всех современных интерпретаторах JavaScript ныне поддерживается метод `forEach()` для обработки массивов, попробуем создать собственную (упрощенную) версию такого метода в виде функции, демонстрируемой в примере кода из листинга 4.12.

Листинг 4.12. Создание функции `forEach()` с целью продемонстрировать установку контекста функции

```
function forEach(list, callback) {
  for (var n = 0; n < list.length; n++) {
    callback.call(list[n], n);
  }
}
var weapons = [ { type: 'shuriken' },
                { type: 'katana' },
                { type:'nunchucks' }];

forEach(weapons, function(index){
  assert(this === weapons[index],
         "Got the expected value of " + weapons[index].type);
});
```

Итеративная функция, которой передается перебираемая коллекция и функция обратного вызова

При каждом обратном вызове в качестве контекста устанавливается текущий перебираемый элемент

Субъект тестирования

Вызвать итеративную функцию и убедиться в правильности ее контекста при каждом обратном вызове

Рассматриваемая здесь итеративная функция отличается простой сигнатурой. В качестве первого аргумента ей передается массив объектов для циклического обращения к ним, а в качестве второго аргумента — функция обратного вызова. В функции `forEach()` организуется циклическое обращение к элементам массива, в ходе которого для каждого элемента массива вызывается функция обратного вызова, как показано ниже.

```
function forEach(list,callback) {
  for (var n = 0; n < list.length; n++) {
```

```
    callback.call(list[n], n);
  }
}
```

В данном случае используется метод `call()` функции обратного вызова, которому передается текущий элемент массива в качестве первого аргумента, а параметр цикла (он же индекс итерации) — в качестве второго аргумента. Это *должно* привести к тому, что текущий элемент массива становится контекстом функции, а индекс итерации передается функции обратного вызова в качестве единственного аргумента.

Далее выполняется тест, приведенный ниже. С этой целью сначала подготавливается простой массив, а затем вызывается функция `forEach()`, которой передается тестируемый массив и функция обратного вызова, где, собственно, и проверяется, устанавливается ли предполагаемый элемент массива в качестве контекста функции всякий раз, когда вызывается функция обратного вызова. Как показано на рис. 4.5, функция `forEach()` работает безупречно.

```
forEach(weapons, function(index){
  assert(this === weapons[index],
         "Got the expected value of " + weapons[index].type);
});
```

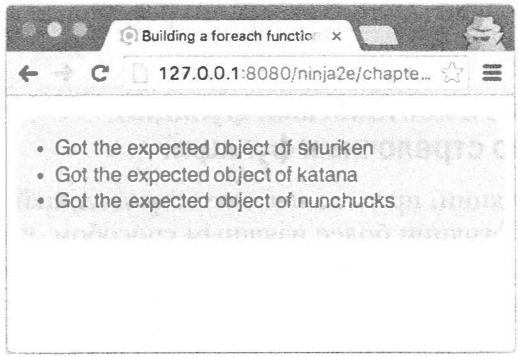


Рис. 4.5. Как показывают результаты тестирования, любой избранный объект можно сделать контекстом функции обратного вызова

Для окончательной реализации подобной функции предстоит еще немало сделать. В частности, решить следующие вопросы: что, если первый аргумент не является массивом, а второй — функцией, и как предоставить автору вебстраницы возможность прервать цикл в любой момент? В качестве упражнения можете усовершенствовать данную функцию, разрешив эти вопросы. А в качестве еще одного упражнения дополните ее возможностью передавать произвольное число аргументов функции обратного вызова, помимо индекса итерации.

Но если методы `apply()` и `call()` делают практически одно и то же, то как же выбрать среди них наиболее подходящий для применения? Общий ответ на этот вопрос такой же, как и на многие подобные вопросы: применять следует тот метод, который повышает ясность кода. А более практический ответ состоит в следующем: применять следует тот метод, который делает более удобным обращение с аргументами. Так, если имеется ряд не связанных друг с другом значений, заданных в переменных или литералах, метод `call()` позволяет указать их непосредственно в списке своих аргументов. Но если значения аргументов уже находятся в массиве или же если их удобнее собрать в массив, то лучше выбрать метод `apply()`.

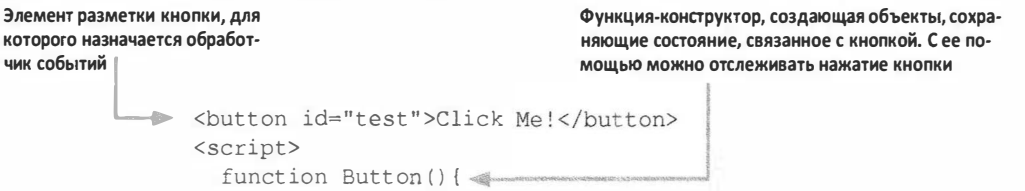
4.3. Разрешение затруднений, связанных с контекстами функций

В предыдущем разделе были показаны затруднения, которые могут возникать при обращении с контекстом функции в JavaScript. В частности, контекст функций обратного вызова (например, обработчиков событий) может оказаться не таким, как предполагалось, но такое затруднение можно разрешить с помощью методов `call()` и `apply()`. В этом разделе будут представлены две другие возможности: стрелочные функции и метод `bind()`, которые в ряде случаев позволяют добиться того же самого результата, но намного более изящным способом.

4.3.1. Обращение с контекстами функций с помощью стрелочных функций

Стрелочные функции, представленные в предыдущей главе, не только позволяют создавать функции более изящным способом, чем объявления функций и функциональные выражения, но и особенно пригодны в качестве функций обратного вызова, поскольку у них отсутствует собственное значение параметра `this`. Вместо этого они запоминают значение параметра `this` в момент своего определения. Вернемся к трудностям, возникающим при обратных вызовах при нажатии экранной кнопки в рассмотренном ранее примере. В коде из листинга 4.13 демонстрируется, каким образом подобные трудности преодолеваются с помощью стрелочных функций.

Листинг 4.13. Обращение с контекстами функций с помощью стрелочных функций



```
    this.clicked = false;
    this.click = () => {
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");//
    };
  }
  var button = new Button();
  var elem = document.getElementById("test");
  elem.addEventListener("click", button.click);
</script>
```

Объявить стрелочную функцию в качестве обработчика событий. А поскольку это метод объекта, то в теле функции можно использовать параметр `this` для получения ссылки на объект

В теле метода проверяется, насколько верно изменилось состояние экранной кнопки после щелчка на ней

Установить обработчик событий от щелчков на экранной кнопке

Единственное изменение в примере кода из листинга 4.13 по сравнению с кодом из листинга 4.10 заключается в использовании приведенной ниже стрелочной функции.

```
this.click = () => {
  this.clicked = true;
  assert(button.clicked, "The button has been clicked");
};
```

Если выполнить код из листинга 4.13, то на экран будет выведен результат, приведенный на рис. 4.6.

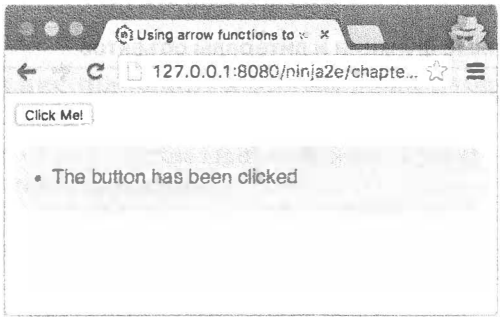


Рис. 4.6. У стрелочных функций отсутствует собственный контекст. Вместо этого они наследуют контекст из той функции, в которой определяются. В частности, параметр **`this`** в стрелочной функции обратного вызова содержит ссылку на объект **`button`**

Как видите, код из рассматриваемого здесь примера вполне работоспособен. В частности, объект `button` отслеживает состояние `clicked` нажатия экранной кнопки. Но в данном случае обработчик событий от щелчков на экранной кнопке создан в конструкторе объектов типа `Button` в виде стрелочной функции:

```
function Button(){
  this.clicked = false;
  this.click = () => {
    this.clicked = true;
```

```
    assert(button.clicked, "The button has been clicked");
  };
}
```

Как упоминалось ранее, стрелочные функции не получают свой неявный параметр `this`, когда они вызываются. Вместо этого они запоминают значение параметра `this` в тот момент, когда создаются. В данном примере кода стрелочная функция `click()` была создана в теле функции-конструктора, где параметр `this` обозначает вновь созданный объект. Поэтому всякий раз, когда мы (или браузер) вызываем стрелочную функцию `click()`, значение параметра `this` постоянно привязано к вновь созданному объекту `button`.

Разъяснение по поводу стрелочных функций и литералов объектов

Значение параметра `this` выбирается в тот момент, когда создается стрелочная функция, что может стать причиной странного, на первый взгляд, поведения. Для разъяснения вернемся к рассмотренному ранее примеру обработчика событий от щелчков на экранной кнопке. Допустим, мы пришли к следующему выводу: нам требуется функция-конструктор, поскольку у нас имеется лишь одна кнопка. Заменим функцию-конструктор простым литералом объекта, как демонстрируется в примере кода из листинга 4.14.

Листинг 4.14. Стрелочные функции и литералы объектов

```
<button id="test">Click Me!</button>
<script>
  assert(this === window, "this === window");
  var button = {
    clicked: false,
    click: () => {
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
      assert(this === window, "In arrow function this === window");
      assert(window.clicked, "clicked is stored in window");
    }
  }

  var elem = document.getElementById("test");
  elem.addEventListener("click", button.click);
</script>
```

Объект `button` определяется в виде литерала объекта

Значением параметра `this` в глобальном коде является глобальный объект `window`

Стрелочная функция является свойством литерала объекта

Проверить, была ли нажата кнопка

Состояние нажатия кнопки сохраняется в окне

Значением параметра `this` в стрелочной функции является глобальный объект `window`

Если мы выполним код из листинга 4.14, нас снова постигнет разочарование, поскольку объекту `button` опять не удалось отследить состояние `clicked` (рис. 4.7).

Правда, в рассматриваемом здесь примере кода мы предусмотрительно разместили ряд утверждений, которые помогут нам выяснить, что же происходит

на самом деле. В частности, следующее утверждение мы разместили непосредственно в глобальном коде, чтобы проверить значение параметра this:

```
assert(this === window, "this === window");
```

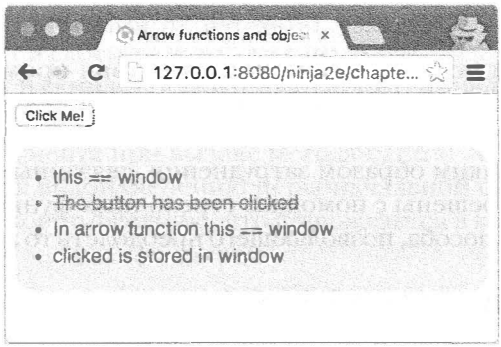


Рис. 4.7. Если стрелочная функция определяется в литерале объекта, который, в свою очередь, определяется в глобальном коде, то значением параметра **this**, связанного со стрелочной функцией, оказывается глобальный объект **window**

Данное утверждение проходит, а это позволяет убедиться в том, что в глобальном коде параметр this ссылается на глобальный объект window. В соответствии с этим мы определяем в литерале объекта button свойство click в виде стрелочной функции:

```
var button = {
  clicked: false,
  click: () => {
    this.clicked = true;
    assert(button.clicked,"The button has been clicked");
    assert(this === window, "In arrow function this === window");
    assert(window.clicked, "Clicked is stored in window");
  };
}
```

А теперь напомним следующее небольшое, но очень важное правило: *стрелочные функции получают значение параметра* this *в момент их создания*. Стрелочная функция click() создается в виде значения свойства литерала объекта, а литерал объекта — в глобальном коде, и поэтому параметр this этой стрелочной функции приобретает то значение, которое он получает в глобальном коде. И как следует из приведенного ниже первого утверждения, размещаемого в глобальном коде, значением параметра this в глобальном коде оказывается глобальный объект window.

```
assert(this === window, "this === window");
```

Таким образом, свойство `clicked` будет определено в глобальном объекте `window`, а не в объекте `button`. Чтобы убедиться в этом, мы проверяем в самом конце рассматриваемого здесь примера кода, что объекту `window` действительно присвоено свойство `clicked`:

```
assert(window.clicked, "Clicked is stored in window");
```

Как видите, если не учитывать все последствия применения стрелочных функций, это может привести к появлению трудно выявляемых программных ошибок. Поэтому будьте начеку!

Итак, выяснив, каким образом затруднения, связанные с контекстом функций, могут быть разрешены с помощью стрелочных функций, перейдем к рассмотрению другого способа, позволяющего преодолеть то же самое затруднение.

4.3.2. Применение метода `bind()`

В примерах кода, рассмотренных ранее в этой главе, уже встречались два метода, `call()` и `apply()`, доступных каждой функции. А кроме того, было показано, как пользоваться этими методами для получения большего контроля над контекстом функций и аргументами, передаваемыми при их вызове.

Помимо этих методов, каждой функции доступен метод `bind()`, который служит для создания новой функции. Эта функция имеет то же самое тело, но ее контекст *всегда* привязан к определенному объекту *независимо* от способа ее вызова. Рассмотрим снова (и в последний раз) незначительное затруднение, возникшее в предыдущем примере в связи с обработчиками событий от щелчков на экранной кнопке, как демонстрируется в примере кода из листинга 4.15.

Листинг 4.15. Привязка конкретного контекста к обработчику событий

```
<button id="test">Click Me!</button>
<script>
  var button = {
    clicked: false,
    click: function(){
      this.clicked = true;
      assert(button.clicked,"The button has been clicked");
    }
  };
  var elem = document.getElementById("test");
  elem.addEventListener("click", button.click.bind(button));

  var boundFunction = button.click.bind(button);
  assert(boundFunction != button.click,
         "Calling bind creates a completly new function");
</script>
```

Использовать метод `bind()` для создания новой функции, привязываемой к объекту `button`

Секретным дополнением в коде из листинга 4.15 служит приведенный ниже метод `bind()`.

```
elem.addEventListener("click", button.click.bind(button));
```

Метод `bind()` доступен для всех функций и предназначен для *создания* и возврата *новой* функции, привязанной к объекту, передаваемому этому методу (в данном случае — к объекту `button`). Значение параметра `this` всегда устанавливается равным этому объекту независимо от способа вызова функции. Помимо этого, привязанная функция ведет себя как исходная функция, поскольку она содержит в своем теле тот же самый код.

Всякий раз, когда нажимается экранная кнопка, привязанная функция вызывается с объектом `button` в качестве ее контекста, поскольку объект `button` указан в качестве аргумента при вызове метода `bind()`. Однако вызов метода `bind()` не приводит к видоизменению первоначальной функции. Этот метод создает совершенно новую функцию, что утверждается в конце кода из данного примера:

```
var boundFunction = button.click.bind(button);
assert(boundFunction != button.click,
       "Calling bind creates a completly new function");
```

На этом исследование контекста функции завершается. Сделайте небольшой перерыв, поскольку в следующей главе вам придется ознакомиться с замыканиями — одним из самых важных и сложных понятий в JavaScript.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- При вызове функции, помимо параметров, явно указываемых в определении функции, передаются два неявных параметра, `arguments` и `this`, имеющие следующее назначение:
  - Параметр `arguments` содержит коллекцию аргументов, передаваемых функции. У него имеется свойство `length`, обозначающее количество переданных аргументов и обеспечивающее доступ к значениям тех аргументов, для которых отсутствуют соответствующие параметры. В нестрогом режиме объект `arguments` назначает псевдонимы параметрам функции (при изменении значения аргумента изменяется значение параметра, и наоборот). Но этого можно избежать, используя строгий режим.
  - Параметр `this` представляет контекст функции, т.е. объект, к которому привязывается вызов функции. Порядок определения параметра `this` может зависеть от того, каким образом определяется и вызывается функция.
- Функцию можно вызвать следующими способами.
  - Как функцию, например `skulk()`.
  - Как метод, например `ninja.skulk()`.
  - Как конструктор, например `new Ninja()`.

  - Через методы `apply()` и `call()`, например `skulk.call(ninja)` или `skulk.apply(ninja)`.

- Способ вызова функции оказывает влияние на значение параметра `this` следующим образом.
  - Если функция вызывается как функция, значением параметра `this` обычно становится глобальный объект `window` в нестрогом режиме, а в строгом режиме — неопределенное значение `undefined`.
  - Если функция вызывается как метод, значением параметра `this` обычно становится тот объект, для которого вызывается данная функция.
  - Если функция вызывается как конструктор, значением параметра `this` становится вновь созданный объект.
  - Если функция вызывается через методы `call()` и `apply()`, значением параметра `this` становится значение первого аргумента, передаваемого методу `call()` или `apply()`.
- У стрелочных функций отсутствует собственное значение параметра `this`. Вместо этого они выбирают значение данного параметра в момент своего создания.
- Метод `bind()`, доступный всем функциям, служит для создания новой функции, которая всегда привязана к аргументу данного метода. А в остальном новая функция ведет себя таким же образом, как и исходная.

Упражнения

1. Следующая функция вычисляет сумму передаваемых ей аргументов, используя для этой цели объект `arguments`:

```
function sum(){
  var sum = 0;
  for(var i = 0; i < arguments.length; i++){
    sum += arguments[i];
  }

  return sum;
}

assert(sum(1, 2, 3) === 6, 'Sum of first three numbers is 6');
assert(sum(1, 2, 3, 4) === 10, 'Sum of first four numbers is 10');
```

   Используя оставшиеся параметры, представленные в предыдущей главе, перепишите функцию `sum()` таким образом, чтобы не пользоваться в ней объектом `arguments`.

2. Какие значения примут переменные `ninja` и `samurai` после выполнения приведенного ниже фрагмента кода?

```
function getSamurai(samurai){
  "use strict"

  arguments[0] = "Ishida";

  return samurai;
}

function getNinja(ninja){
  arguments[0] = "Fuma";
  return ninja;
}

var samurai = getSamurai("Toyotomi");
var ninja = getNinja("Yoshi");
```

3. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
function whoAmI1(){
  "use strict";

  return this;
}

function whoAmI2(){
  return this;
}

assert(whoAmI1() === window, "Window?");
assert(whoAmI2() === window, "Window?");
```

4. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
var ninja1 = {
  whoAmI: function(){
    return this;
  }
};

var ninja2 = {
  whoAmI: ninja1.whoAmI
};

var identify = ninja2.whoAmI;

assert(ninja1.whoAmI() === ninja1, "ninja1?");
assert(ninja2.whoAmI() === ninja1, " ninja1 again?");

assert(identify() === ninja1, "ninja1 again?");
assert(ninja1.whoAmI.call(ninja2) === ninja2, "ninja2 here?");
```

5. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
function Ninja(){
  this.whoAmI = () => this;
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};

assert(ninja1.whoAmI() === ninja1, "ninja1 here?");
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

6. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
function Ninja(){
  this.whoAmI = function(){
    return this;
  }.bind(this);
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};

assert(ninja1.whoAmI() === ninja1, "ninja1 here?");
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

5 Функции для мастера: замыкания и области видимости

В этой главе...

- Применение замыканий для упрощения разработки веб-приложений
- Отслеживание работы программ на JavaScript с помощью контекстов выполнения
- Отслеживание областей видимости переменных с помощью лексических сред
- Представление о типах переменных
- Исследование принципа действия замыканий

Замыкания – это определяющее языковое средство JavaScript, тесно связанное с функциями, подробно рассмотренными в предыдущих главах. Несмотря на то что многие разработчики приложений на JavaScript пишут код без ясного представления о преимуществах замыканий, последние не только помогают сократить объем и уменьшить сложность прикладного кода, требующегося для внедрения более развитых средств, но и позволяют делать то, что просто невозможно или слишком сложно осуществить без них. Например, решить любые задачи, связанные с обратными вызовами, включая обработку событий или анимацию, было бы значительно труднее без замыканий, а другие задачи вроде поддержки закрытых объектных переменных были бы просто неосуществимы. Внедрение замыканий и способы написания с их помощью кода окончательно определило перспективу JavaScript как языка программирования.

По традиции замыкания всегда считались исключительно средством языков функционального программирования, и поэтому особенно приветствовалось их внедрение в практику массовой разработки приложений. Теперь замыкания проникли во многие библиотеки JavaScript и другие развитые кодовые базы благодаря их способности существенно упрощать сложные операции.

Побочно замыкания оказывают влияние на области видимости в JavaScript. Именно поэтому мы исследуем в этой главе правила соблюдения области видимости, принятые в JavaScript, уделив особое внимание последним дополнениям этого языка. Это поможет вам лучше понять внутренний механизм действия замыканий.

Знаете ли вы?

Сколько областей видимости может быть у переменной или метода и каковы они?

Каким образом отслеживаются идентификаторы и их значения?

Что такое изменяемая переменная и каким образом она определяется в JavaScript?

5.1. Общее представление о замыканиях

Замыкание предоставляет функции доступ к внешним по отношению к ней переменным и позволяет манипулировать ими. Замыкания делают доступными для функции все переменные, а также другие функции, которые оказываются в области видимости во время ее определения.

Примечание

Вы, вероятно, уже знакомы с понятием областей видимости, но на всякий случай поясним, что *область видимости* определяет доступность (или так называемую *видимость*) идентификаторов в определенных частях программы. Область видимости является неотъемлемой частью программы, где определенное имя привязано к конкретной переменной.

На первый взгляд понятие замыкания кажется вполне интуитивным, но не следует забывать, что объявленная функция может быть вызвана в любой последующий момент, даже *после* выхода из области видимости, в которой она была объявлена. Это понятие, вероятно, лучше всего пояснить на примере конкретного кода. Но прежде чем перейти к конкретным примерам, которые помогут вам разрабатывать более изящную анимацию в прикладном коде или определять закрытые свойства объектов, рассмотрим небольшой пример кода из листинга 5.1.

Листинг 5.1. Простое замыкание

```
var outerValue = "ninja";          <-- Определить переменную в глобальной области видимости
function outerFunction(){          <-- Объявить функцию в глобальной области видимости
```

```
  assert(outerValue === "ninja","I can see the ninja.");
}
outerFunction();        ← Запустить функцию
```

В приведенном выше простом примере кода переменная `outerValue` и функция `outerFunction()` объявляются в одной и той же области видимости (в данном случае в глобальной). После этого функция `outerFunction()` выполняется.

Как показано на рис. 5.1, переменная `outerValue` доступна для данной функции. Такой код вам, вероятно, приходилось писать не раз, даже не осознавая, что вы невольно создавали замыкание!

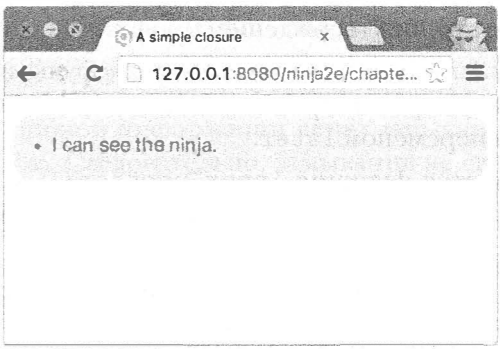


Рис. 5.1. Данная функция обнаружила ниндзя, скрывавшегося в пределах простой видимости

Приведенный выше пример, по-видимому, не особенно впечатляет и даже не удивляет. Переменная `outerValue` и функция `outerFunction()` объявлены в глобальной области видимости, которая вообще не покидается (до тех пор, пока выполняется приложение), и поэтому не удивительно, что переменная существует в данной области видимости и по-прежнему доступна для функции. Но несмотря на наличие замыкания, его преимущества в данном примере пока еще неясны. Усложним немного пример, как показано в листинге 5.2.

Листинг 5.2. Еще один пример замыкания

Объявить переменную в теле функции. Область видимости этой переменной ограничивается телом функции, и поэтому она недоступна за пределами данной функции

```
var outerValue = "samurai";
var later;

function outerFunction(){
  var innerValue = "ninja";
  function innerFunction(){
    assert(outerValue === "samurai", "I can see the samurai.");
    assert(innerValue === "ninja", "I can see the ninja.")
  }

  later = innerFunction;
}
```

Пустая переменная, которая применяется в дальнейшем

Объявить внутреннюю функцию в теле функции `outerFunction()`. Переменная `innerValue` находится в области видимости этой функции при ее создании

Сохранить ссылку на функцию `innerFunction()` в переменной `later`. Эта переменная находится в глобальной области видимости, что дает возможность вызвать в дальнейшем данную функцию

```
outerFunction();
later();
```

Вызвать функцию `outerFunction()`, что приведет к созданию функции `innerFunction()` и присваиванию ссылки на нее переменной `later`

Вызвать функцию `innerFunction()` через переменную `later`. Ее нельзя вызвать непосредственно, так как область ее видимости (а также переменной `innerValue`) не выходит за пределы функции `outerFunction()`

Проанализируем код из функции `innerFunction()` в приведенном выше примере и попробуем спрогнозировать, что при этом может произойти.

- Первое утверждение, безусловно, должно пройти, поскольку переменная `outerValue` находится в глобальной области видимости и доступна повсюду. А что же второе утверждение?
- Функция `innerFunction()` выполняется *после* функции `outerValue outer Function()` благодаря копированию ссылки на нее в глобальную ссылку, сохраняемую в переменой `later`.
- Когда выполняется функция `innerFunction()`, область видимости в функции `outerFunction()` уже покинута и недоступна на момент вызова этой функции по ссылке в переменной `later`.
- Следовательно, можно было бы с полной уверенностью предположить, что второе утверждение не пройдет, поскольку значение переменной `innerValue` не определено (`undefined`), не так ли?

Тем не менее тест проходит успешно, как показано на рис. 5.2.

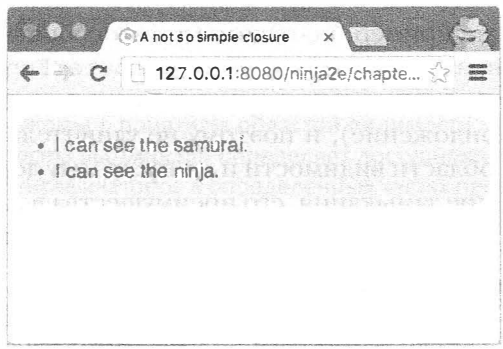


Рис. 5.2. Несмотря на попытку скрыться во внутренней функции, ниндзя обнаружен!

Как же такое возможно? Каким чудом переменная `innerValue` все еще доступна и существует при выполнении внутренней функции после того, как мы уже давно вышли из области видимости, в которой она была создана,? Ответ, разумеется, следует искать в замыканиях.

При объявлении внутренней функции `innerFunction()` во внешней функции было определено ее объявление, а также образовано замыкание, охватывающее не только эту функцию, но и все переменные, находящиеся в об-

ласти ее видимости *на момент объявления* данной функции. Когда же функция `innerFunction()` выполняется, даже *после* фактического выхода из области видимости, в которой она была объявлена, она по-прежнему имеет доступ к этой исходной области видимости через свое замыкание (рис. 5.3).

В этом, собственно, и состоит принцип действия замыканий. Они образуют "защитную оболочку" вокруг функции и переменных, находящихся в области ее видимости на момент объявления данной функции, и благодаря этому у нее имеется все необходимое для выполнения. Эта "оболочка", охватывающая как саму функцию, так и ее переменные, остается до тех пор, пока существует сама функция.

И хотя вся эта сигнатура не сразу видна (ведь не существует объекта замыкания, содержащего всю эту информацию, как можно было бы предположить), тем не менее, такой способ сохранения информации и обращения к ней требует прямых затрат. Не следует, однако, забывать, что на всякую функцию, получающую доступ к подобной информации через замыкание, возложено бремя нести эту информацию. Следовательно, замыкания не свободны от издержек, хотя они и крайне удобны. Ведь всю эту информацию нужно хранить в оперативной памяти до тех пор, пока интерпретатору JavaScript не станет совершенно ясно, что она больше не нужна и ее можно благополучно собрать в "мусор", или же до тех пор, пока не завершится выгрузка страницы.

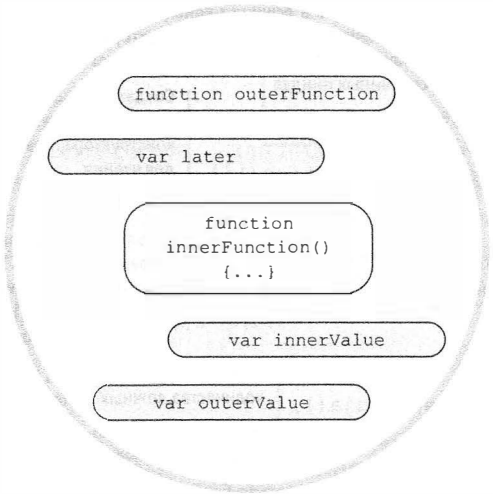


Рис. 5.3. Подобно защитной оболочке, замыкание функции **`innerFunction()`** сохраняет переменные в области ее видимости до тех пор, пока существует сама функция

Впрочем, это еще не все, что следовало бы сказать о внутренних механизмах действия замыканий. Но прежде чем исследовать механизмы, которые активизируют замыкания, рассмотрим примеры их практического применения.

5.2. Применение замыканий на практике

Итак, мы рассмотрели в общих чертах, что собой представляют замыкания и как они действуют. А теперь покажем, как они применяются на практике при разработке приложений на JavaScript. Уделим пока что основное внимание практическим вопросам и выгодам от применения замыканий. Далее в этой главе мы снова рассмотрим те же самые примеры применения замыканий, чтобы выяснить внутренние механизмы их действия.

5.2.1. Имитация закрытых переменных

Во многих языках программирования применяются *закрытые переменные* — свойства объекта, скрытые от внешнего мира. Это языковое средство удобно для того, чтобы не перегружать пользователей объектов ненужными подробностями реализации при доступе к объектам из других частей кода. К сожалению, в JavaScript отсутствует собственная поддержка закрытых переменных. Но, используя понятие замыкания, мы можем все-таки добиться похожего и вполне приемлемого результата, как демонстрируется в примере кода из листинга 5.3.

Листинг 5.3. Приближенное представление закрытых переменных с помощью замыканий

Объявить конструктор объектов типа `Ninja`

Объявить переменную в функции-конструкторе. Область видимости этой переменной ограничивается телом конструктора, поэтому она оказывается "закрытой" переменной. Она послужит для подсчета количества ложных выпадов, сделанных ниндзя

Создать метод доступа к подсчету ложных выпадов. Переменная `feints` недоступна за пределами конструктора, поэтому ее значение доступно только для чтения

Объявить метод для инкрементирования значения переменной feints. Это закрытая переменная, поэтому ее значение никто не может изменить без разрешения. Посторонним предоставляется ограниченный доступ к ней через методы

Протестировать код, создав сначала экземпляр объекта типа `Ninja`

Вызвать метод `feint()`, в котором наращивается подсчет количества ложных выпадов со стороны ниндзя

Проверить, можно ли получить прямой доступ к переменной

Когда новый объект `ninja2` создается с помощью конструктора объектов типа `Ninja`, он получает свою переменную `feints`

"Закрытую" переменную удалось изменить, несмотря на отсутствие прямого доступа к ней

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}

var ninja1 = new Ninja();
ninja1.feint();

assert(ninja1.feints === undefined,
       "And the private data is inaccessible to us.");
assert(ninja1.getFeints()=== 1,
       "We're able to access the internal feint count.");

var ninja2 = new Ninja();
assert(ninja2.getFeints()=== 0,
       "The second ninja object gets its own feints variable.");
```

В примере кода из листинга 5.3 создается функция, служащая в качестве конструктора объектов типа `Ninja`. Применение функции в качестве конструктора рассматривалось в главе 4 и будет продолжено в главе 7. А до тех пор достаточно напомнить, что если указать ключевое слово `new` при обращении к функции, то будет создан экземпляр нового объекта, а функция — вызвана с новым объектом в качестве ее контекста, служа для этого объекта в качестве конструктора. Следовательно, ссылка `this` в теле функции указывает на полученный экземпляр нового объекта.

В самом конструкторе для хранения состояния определяется переменная `feints`. Правила соблюдения области видимости в JavaScript ограничивают доступность этой переменной *в пределах* конструктора. Чтобы получить доступ к значению данной переменной из кода за пределами области ее видимости, в рассматриваемом здесь коде определяется *метод доступа* `getFeints()`, с помощью которого можно только прочитать значение закрытой переменной, как показано ниже. (Методы доступа зачастую называют *методами-получателями* или просто "получателями".)

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}
```

Далее реализуется метод `feint()` для полноценного контроля над значением переменной `feints`. В реальном приложении это может быть какой-нибудь метод, реализующий бизнес-логику, а в данном случае он просто инкрементирует значение переменной `feints`.

Как только конструктор завершит свою работу, для вновь созданного объекта `ninja1` можно вызвать метод `feint()`:

```
var ninja1 = new Ninja();
ninja1.feint();
```

Как показывают тесты в данном примере кода, с помощью метода доступа можно получить *значение* закрытой переменной, но не прямой доступ к ней. Это предотвращает неконтролируемые изменения значения переменной, так как если бы она была полноценной закрытой переменной. Данная ситуация наглядно показана на рис. 5.4.

Применяя замыкания, можно сохранять состояние объекта типа `Ninja` в методе, не предоставляя прямой доступ к этому состоянию пользователю метода. Ведь переменная `feints` доступна для внутренних методов через их замыкания, но не для кода, находящегося за пределами конструктора.

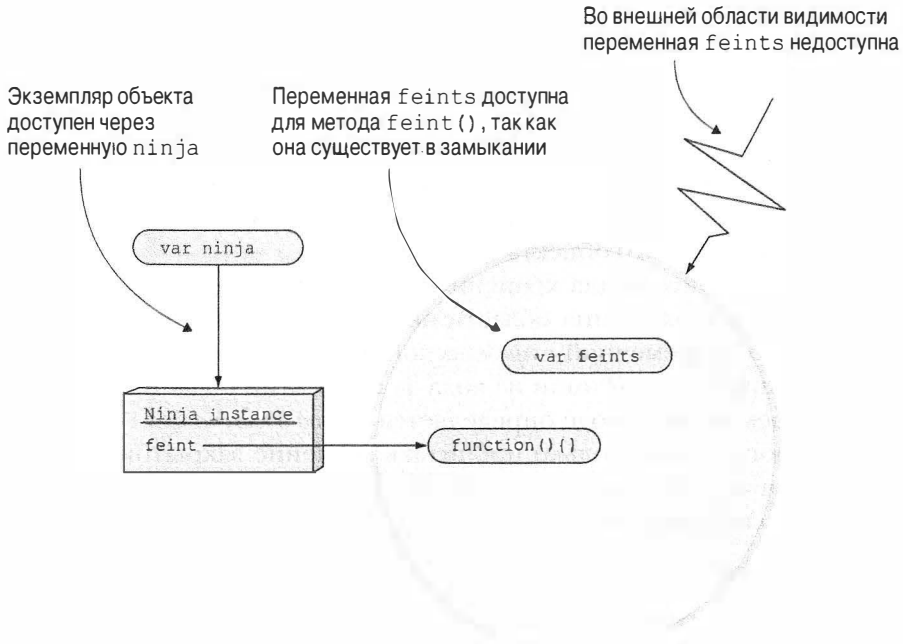


Рис. 5.4. Сокрытие переменной в конструкторе делает ее недоступной во внешней области видимости, но она по-прежнему существует и надежно защищена в замыкании

Данный пример раскрывает объектно-ориентированный характер JavaScript, который будет более углубленно исследован в главе 7. А пока уделим основное внимание другому типичному примеру применения замыканий.

5.2.2. Применение замыканий при обратных вызовах

Еще одно полезное применение замыкания находят, когда приходится иметь дело с обратными вызовами, т.е. в тех случаях, когда функция вызывается в неопределенный в дальнейшем момент времени. Зачастую в подобных функциях требуется доступ к внешним данным. В примере кода из листинга 5.4 демонстрируется создание простой анимации с помощью таймеров и функций обратного вызова.

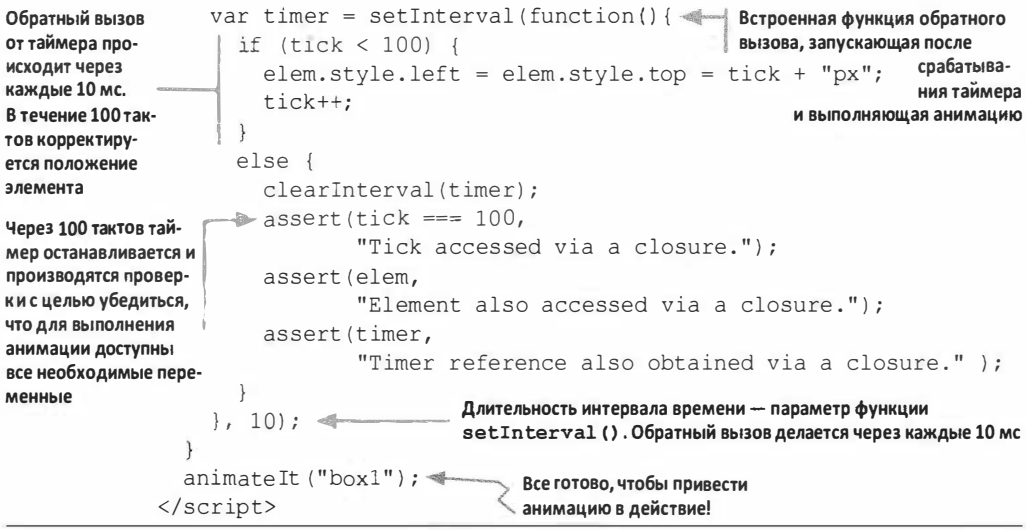
Листинг 5.4. Применение замыкания при обратном вызове через определенный промежуток времени срабатывания таймера

Установить счетчик для отслеживания количества тактов (шагов) анимации

Создать элемент, который требуется подвергнуть анимации

В теле функции `animateIt()` ищется ссылка на этот элемент

```
<div id="box1">First Box</div>
<script>
  function animateIt(elementId) {
    var elem = document.getElementById(elementId);
    var tick = 0;
```




```
var timer = setInterval(function(){
  if (tick < 100) {
    elem.style.left = elem.style.top = tick + "px";
    tick++;
  }
  else {
    clearInterval(timer);
    assert(tick === 100,
           "Tick accessed via a closure.");
    assert(elem,
           "Element also accessed via a closure.");
    assert(timer,
           "Timer reference also obtained via a closure." );
  }
}, 10);
}
animateIt("box1");
</script>
```

Особое внимание следует обратить на применение в коде из листинга 5.4 единственной анонимной функции, указываемой в качестве аргумента функции `setInterval()` для анимации целевого элемента разметки `div`. Для управления процессом анимации в этой функции через замыкание доступны три переменные (`elem`, `tick` и `timer`). Все три переменные (`elem` — для ссылки на элемент модели DOM, `tick` — для подсчета тактов анимации, `timer` — для ссылки на таймер) должны быть доступны на протяжении всего времени анимации и находиться за пределами глобальной области видимости.

Но код из данного примера будет работоспособным и в том случае, если вынести упомянутые выше переменные из функции `animateIt()` в глобальную область видимости. А зачем же тогда поднимается вся эта шумиха о том, чтобы не засорять глобальную область видимости?

Попробуйте вынести эти переменные в глобальную область видимости и убедиться, что код из данного примера остается по-прежнему работоспособным. Затем видоизмените этот код, чтобы осуществить анимацию двух элементов, введя еще один элемент с уникальным идентификатором и вызвав функцию `animateIt()` с этим идентификатором сразу же после вызова первой функции.

И тотчас обнаружится следующее затруднение: если хранить переменные в глобальной области видимости, то для *каждой* анимации потребуются три отдельные переменные. В противном случае они будут перескакивать друг через друга, пытаясь использовать один и тот же ряд переменных для отслеживания многих состояний.

Определяя переменные в *теле* функции и используя замыкания, чтобы сделать их доступными для функции обратного вызова таймера, каждая анимация получает свой "закрытый" набор переменных (рис. 5.5).

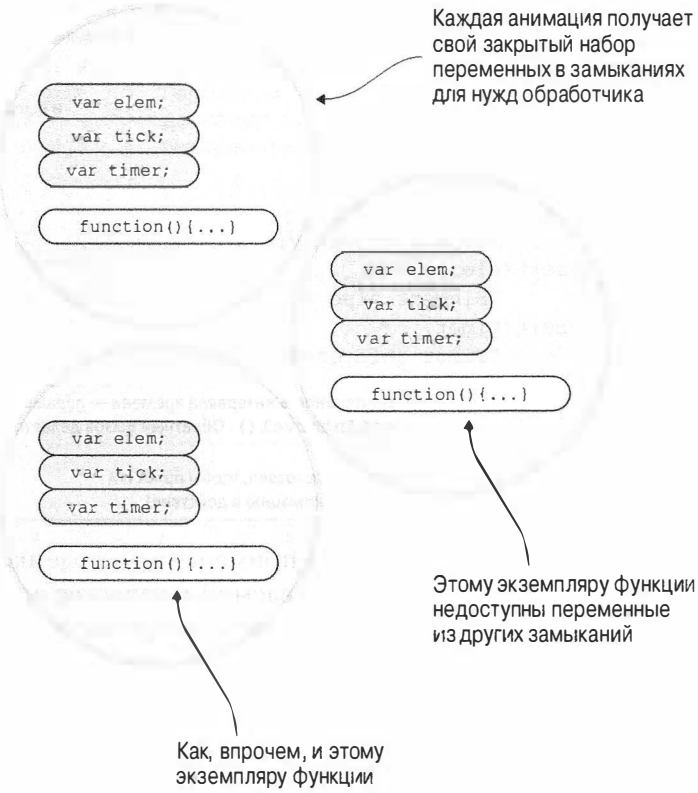


Рис. 5.5. Сохраняя переменные отдельно для нескольких экземпляров функции, можно выполнять одновременно несколько операций анимации

Без замыканий выполнять одновременно несколько операций, будь то обработка событий, анимация или даже составление запросов к серверу, крайне трудно. И это служит веским основанием для досконального овладения замыканиями!

Рассматриваемый здесь пример особенно наглядно демонстрирует способность замыканий производить удивительно интуитивный и лаконичный код. Введя переменные в функцию `animateIt()`, мы тем самым создали в данном примере неявное замыкание, не прибегая к сложному синтаксису.

Данный пример проясняет еще одно важное понятие. Нам доступны не только значения переменных в момент создания замыкания, но и их обновление в замыкании в то время, как в нем выполняется функция. Замыкание является не моментальным снимком состояния области видимости в момент ее создания, а активной инкапсуляцией этого состояния, которое можно видоизменить при условии, что замыкание существует.

Замыкания тесно связаны с областями видимости, поэтому мы посвятим немалую долю остальной части этой главы исследованию правил соблюдения об-

ласти видимости в JavaScript. Но прежде следует подробно рассмотреть, каким образом выполнение кода отслеживается в JavaScript.

5.3. Отслеживание выполнения кода с помощью контекстов выполнения

Основным исполняемым блоком в JavaScript служит функция. Мы пользуемся функциями постоянно, чтобы вычислить что-нибудь, произвести побочные эффекты вроде изменения пользовательского интерфейса, добиться повторного использования кода или сделать его более удобочитаемым. Чтобы выполнить свое назначение, одна функция может вызвать другую функцию, а та, в свою очередь, еще одну и т.д. А когда функция завершит свою работу, выполнение программы должно вернуться в то место, откуда была вызвана функция. Но задумывались ли вы над тем, каким образом интерпретатор JavaScript отслеживает все выполняемые функции и возвраты в места их вызова?

Как упоминалось в главе 2, в JavaScript имеются две разновидности кода: *глобальный код*, находящийся за пределами всех функций, а также *код функции*, содержащийся в самих функциях. Когда прикладной код выполняется интерпретатором JavaScript, каждый его оператор выполняется в определенном *контексте выполнения*.

И аналогично двум упомянутым выше разновидностям кода, имеются две разновидности контекста выполнения: *глобальный контекст выполнения* и *контекст выполнения функции*. Они существенно отличаются тем, что существует лишь *один* глобальный контекст выполнения, создаваемый при запуске программы на JavaScript, тогда как *новый* контекст выполнения функции создается при *каждом* вызове функции.

Примечание

Как упоминалось в главе 4, *контекст функции* представляет собой объект, для которого она вызывается. Он может быть доступным через параметр, обозначаемый ключевым словом `this`. Но совсем другое дело — *контекст выполнения функции*, хотя он и имеет похожее название. Он используется в интерпретаторе JavaScript для слежения за ходом выполнения функции.

И как упоминалось в главе 2, в основу языка JavaScript положена модель однопоточного выполнения команд процессора, в соответствии с которой одновременно может быть выполнен только один фрагмент кода. Всякий раз, когда функция вызывается, ее текущий контекст выполнения прерывается и создается новый контекст, в котором функция выполняет свое задание. И как только она завершит свою работу, ее контекст выполнения, как правило, удаляется и восстанавливается предыдущий контекст выполнения вызывающего кода. В связи с этим возникает потребность отслеживать все эти контексты выполнения: как тот, который выполняется, так и те, которые терпеливо ожидают сво-

ей очереди. Организовать такое отслеживание проще всего с помощью стека, называемого *стеком контекстов выполнения*, а зачастую — просто *стеком вызовов*.

Примечание

Стек является основополагающей структурой данных, где новые элементы можно размещать только на вершине, а существующие элементы извлекать только из вершины. Стек можно представить в виде стопки подносов в столовой. Если обедающему требуется поднос, он берет его сверху стопки, а работник столовой кладет чистый поднос сверху стопки.

Чтобы прояснить дело, рассмотрим пример кода из листинга 5.5, где сообщается о действиях скрытных ниндзя.

Листинг 5.5. Создание контекстов выполнения

```
function skulk(ninja) {
  report(ninja + " skulking");
}

function report(message) {
  console.log(message);
}

skulk("Kuma");
skulk("Yoshi");
```

Функция, вызывающая другую функцию

Функция, выводящая сообщение через встроенный метод `console.log()`

Два вызова функции и з глобального кода

Код из примера в листинге 5.5 довольно прост. В нем определяется функция `skulk()`, вызывающая функцию `report()`, которая выводит сообщение. А затем в глобальном коде делаются два отдельных вызова функции `skulk()`: `skulk("Kuma")` и `skulk("Yoshi")`. Воспользовавшись кодом из данного примера в качестве основы, исследуем создание контекстов выполнения, как показано на рис. 5.6.

При выполнении кода из рассматриваемого здесь примера контекст выполнения ведет себя следующим образом.

1. Сначала в стеке контекстов выполнения размещается глобальный контекст выполнения, который создается только один раз для каждой программы на JavaScript (или для каждой веб-страницы, если речь идет о браузерах). Глобальный контекст выполнения служит активным контекстом при выполнении глобального кода.
2. В глобальном коде данной программы сначала определяются две функции, `skulk()` и `report()`, а затем вызывается функция `skulk("Kuma")`. Но поскольку одновременно может быть выполнен только один фрагмент кода, интерпретатор JavaScript прерывает выполнение глобального кода и переходит к выполнению кода функции `skulk()` с аргументом `"Kuma"`. С этой целью создается *новый* контекст выполнения функции, который размещается на вершине стека.

3. В свою очередь, функция `skulk()` вызывает функцию `report()` с аргументом `"Kuma skulking"`. И снова контекст выполнения функции `skulk()` прерывается, поскольку одновременно может быть выполнен только один фрагмент кода JavaScript, а далее создается новый контекст выполнения функции `report()` с аргументом `"Kuma skulking"`, который размещается на вершине стека.
4. Как только функция `report()` выведет сообщение с помощью встроенного метода `console.log()` (см. приложение Б к данной книге) и завершит свое выполнение, необходимо вернуться к функции `skulk()`. С этой целью контекст выполнения функции `report()` удаляется из стека, а затем восстанавливается контекст выполнения функции `skulk()` и далее продолжается ее выполнение.

Рис. 5.6. Поведение стека контекстов выполнения

5. Аналогичная последовательность действий происходит, когда функция `skulk()` завершает свое выполнение. В этом случае контекст выполнения функции `skulk()` удаляется из стека, а затем восстанавливается глобальный контекст выполнения, терпеливо ожидающий своей очереди. Таким образом, возобновляется выполнение глобального кода JavaScript.

Весь этот процесс повторяется при втором вызове функции `skulk()`, но на этот раз с аргументом `"Yoshi"`. При этом в стеке создаются и размещаются два новых контекста выполнения функций, когда последовательно делаются вызовы `skulk("Yoshi")` и `report("Yoshi skulking")`. Эти контексты выполнения также удаляются из стека, когда программа возвращается из соответствующей функции.

Несмотря на то что стек контекстов выполнения является внутренним механизмом работы JavaScript, им можно пользоваться в любом отладчике кода JavaScript, где он называется *стеком вызовов*. В качестве примера на рис. 5.7 показан стек вызовов в отладчике Chrome DevTools.

Рис. 5.7. Текущее состояние стека контекстов выполнения в Chrome DevTools

Примечание

Инструментальные средства отладки, доступные в различных браузерах, подробно рассматриваются в приложении Б.

Помимо слежения за местом в выполняемом приложении, контекст выполнения играет важную роль в *анализе идентификаторов* — процессе сопоставления переменных конкретным идентификаторам. В контексте выполнения это делается через *лексическую среду*.

5.4. Отслеживание идентификаторов с помощью лексических сред

Лексическая среда — это внутренняя структура интерпретатора JavaScript, предназначенная для отслеживания процесса сопоставления идентифика-

торов имен конкретным переменным. Рассмотрим в качестве примера приведенный ниже фрагмент кода. Обращение за справкой к лексической среде происходит при доступе к переменной `ninja` в операторе вызова встроенного метода `console.log()`.

```
var ninja = "Hattori";
console.log(ninja);
```

Примечание

Лексические среды являются внутренней реализацией механизма областей видимости интерпретатора JavaScript, поэтому их зачастую просто называют *областями видимости*.

Как правило, лексическая среда связана с конкретной структурой кода JavaScript. Она может быть связана с функцией, блоком кода или частью `catch` оператора `try-catch`. У каждой из этих структур (функций, блоков и частей `catch` операторов `try-catch`) имеются собственные изолированные среды сопоставления идентификаторов.

Примечание

До появления стандарта ES6 языка JavaScript лексическую среду можно было связать только с функцией. Переменные можно было ограничивать только областью видимости функции, и это вызывало немало недоразумений. Но поскольку JavaScript является C-подобным языком, то те разработчики, у которых имеется опыт программирования на C++, C# или Java, естественно, предполагают, что некоторые низкоуровневые понятия (например, области видимости блоков) окажутся в JavaScript такими же, как и в других C-подобных языках. Этот недостаток был наконец-то устранен в стандарте ES6.

5.4.1. Вложение кода

Лексические среды в высокой степени зависят от *вложения кода*, которое дает возможность вкладывать одну структуру кода в другую. Различные типы вложения кода показаны на рис. 5.8.

В данном примере обнаруживается следующее.

- Цикл `for` вложен в функцию `report()`.
- Функция `report()` вложена в функцию `skulk()`.
- Функция `skulk()` вложена в глобальный код.

С точки зрения областей видимости каждая из этих структур кода оказывается связанной с лексической средой *всякий* раз, когда интерпретируется код. Например, при каждом вызове функции `skulk()` создается ее новая лексическая среда.

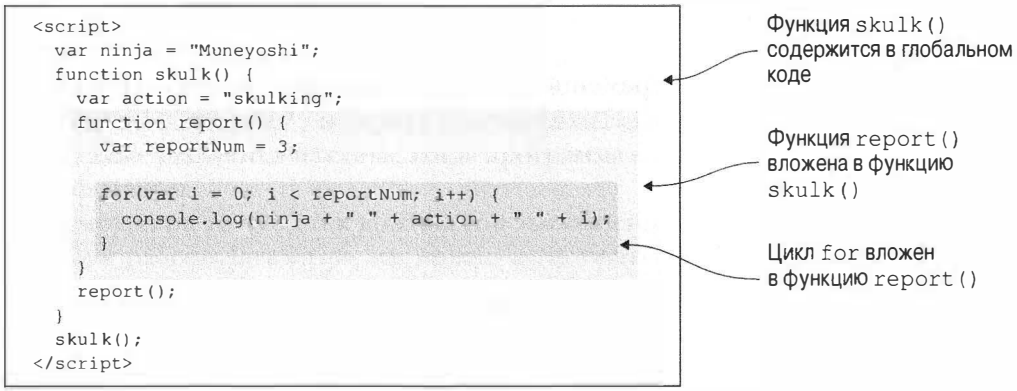


Рис. 5.8. Типы вложения кода

Следует также подчеркнуть, что внутренняя структура кода имеет доступ к переменным, определенным во внешних структурах кода. Например, в цикле `for` могут быть доступны переменные из функции `report()`, функции `skulk()` и глобального кода, а в функции `skulk()` — только дополнительные переменные из глобального кода.

В таком способе доступа к переменным нет ничего особенного. Ведь всем нам приходилось делать это много раз. Но каким образом интерпретатор JavaScript отслеживает все эти переменные и что вообще может быть доступно из конкретного контекста? Именно здесь и приходят на помощь лексические среды.

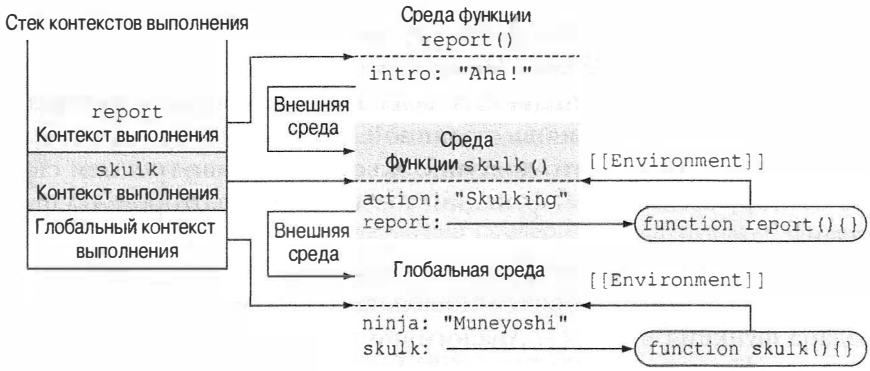
5.4.2. Вложение кода и лексические среды

Помимо отслеживания локальных переменных, объявлений и параметров функций, в каждой лексической среде должна отслеживаться ее *внешняя* (родительская) лексическая среда. Это требуется потому, что необходим доступ к переменным, определяемым во внешних структурах кода. Если идентификатор не удается найти в текущей среде, поиск продолжается во внешней среде. Поиск останавливается, когда обнаруживается искомая переменная или возникает ссылочная ошибка, если достигнута глобальная среда и искомый идентификатор так и не был найден. В примере кода, приведенном на рис. 5.9, показано, как определяются значения идентификаторов `intro`, `action` и `ninja` при выполнении функции `report()`.

В данном примере кода функция `report()` вызывается из функции `skulk()`, которая, в свою очередь, вызывается из глобального кода. С каждым контекстом выполнения связана отдельная лексическая среда, содержащая таблицу всех идентификаторов, определенных непосредственно в данном контексте. Например, в таблице идентификаторов глобальной среды содержится информация о переменных `ninja` и `skulk`; в таблице идентификаторов функции `skulk()` — информация о переменных `action` и `report`, а в таблице идентифи-

каторов функции `report()` — информация о переменной `intro` (см. рис. 5.9, *справа*).

```
<script>
  var ninja = "Muneyoshi";

  function skulk() {
    var action = "Skulking";

    function report() {
      var intro = "Aha!";
      assert(intro === "Aha!", "Local");
      assert(action === "Skulking", "Outer");
      assert(ninja === "Muneyoshi", "Global");
    }

    report();
  }

  skulk();
</script>
```



Найти переменную `intro`

1. Проверить среду функции `report()` -> найдено

Найти переменную `action`

1. Проверить среду функции `report()` -> не найдено
2. Проверить внешнюю среду функции `report()`: функция `skulk()`
   Проверить среду функции `skulk()` -> найдено

Найти переменную `ninja`

1. Проверить среду функции `report()` -> не найдено
2. Проверить внешнюю среду функции `report()`: функция `skulk()`
   Проверить среду функции `skulk()` -> не найдено
3. Проверить внешнюю среду функции `skulk()`: глобальная среда
   Проверить глобальную среду -> найдено

Рис. 5.9. Порядок определения значений переменных в интерпретаторе JavaScript

В конкретном контексте выполнения, помимо доступа к идентификаторам, определяемым непосредственно в соответствующей лексической среде, прикладным программам нередко доступны другие переменные, определенные во внешних средах. Например, в теле функции `report()` требуется доступ к переменной `action` из внешней функции `skulk()`, а также к глобальной переменной `ninja`. С этой целью нужно каким-то образом отслеживать указанные

внешние среды. В JavaScript для этой цели используется тот факт, что функции — это объекты высшего порядка.

Всякий раз, когда создается функция, ссылка на лексическую среду, в которой эта функция была создана, сохраняется во внутреннем (т.е. недоступном для непосредственного манипулирования) свойстве `[[Environment]]` (внутренние свойства принято обозначать двойными квадратными скобками). В данном случае функция `skulk()` будет хранить ссылку на глобальную среду, а функция `report()` — ссылку на среду функции `skulk()`, поскольку именно в этих средах были созданы обе указанные функции.

Примечание

На первый взгляд не совсем понятно, зачем обходить весь стек контекстов выполнения и искать в них соответствующие идентификаторы? Формально это будет работать в рассматриваемом нами примере. Но не следует забывать, что функцию в JavaScript можно передавать как и любой другой объект, и поэтому места, где определяется и вызывается функция, в общем, не связаны (достаточно вспомнить о замыканиях).

Всякий раз, когда вызывается функция, для нее создается новый контекст выполнения и размещается в стеке. Помимо этого, создается новая лексическая среда, связанная с данной функцией. А далее начинается самое интересное: в качестве внешней среды для вновь созданной лексической среды интерпретатор JavaScript выбирает то, что доступно по ссылке во внутреннем свойстве `[[Environment]]` вызываемой функции, т.е. ту среду, в которой эта функция была создана!

В данном случае при вызове функции `skulk()` внешней для вновь созданной среды этой функции становится глобальная среда, поскольку именно в ней была создана функция `skulk()`. Аналогично при вызове функции `report()` внешней для вновь созданной среды этой функции становится среда функции `skulk()`.

А теперь рассмотрим функцию `report()`, которая определяется следующим образом:

```
function report() {
  var intro = "Aha!";
  assert(intro === "Aha!", "Local");
  assert(action === "Skulking", "Outer");
  assert(ninja === "Muneyoshi", "Global");
}
```

При интерпретации первого оператора `assert` в теле данной функции требуется определить, чему соответствует идентификатор переменной `intro`. С этой целью интерпретатор JavaScript начинает проверять среду текущего контекста выполнения, т.е. среду функции `report()`. А поскольку среда функции `report()` содержит ссылку на переменную `intro`, то ситуация с поиском идентификатора благополучно разрешается.

В следующем далее операторе `assert` требуется найти идентификатор переменной `action`. И в этом случае сначала проверяется среда текущего контекста выполнения. Но ведь среда функции `report()` не содержит ссылки на идентификатор переменной `action`, и поэтому интерпретатору JavaScript приходится проверять среду функции `skulk()`, которая оказывается внешней для среды функции `report()`. К счастью, среда функции `skulk()` содержит ссылку на идентификатор переменной `action`, и поэтому ситуация разрешается. Аналогичный процесс происходит и при попытке найти идентификатор переменной `ninja`, который будет обнаружен в глобальной среде.

А теперь, когда стали понятны основы поиска идентификаторов, перейдем к рассмотрению различных способов объявления переменных.

5.5. Общее представление о типах переменных в JavaScript

Для определения переменных в JavaScript служат следующие ключевые слова: `var`, `let`, `const`. Они отличаются двумя особенностями: *изменяемостью* и взаимосвязью с лексической средой.

Примечание

Ключевое слово **`var`** было неотъемлемой частью языка JavaScript с самого начала его существования, тогда как ключевые слова **`let`** и **`const`** появились в только стандарте ES6. Наличие поддержки ключевых слов **`let`** и **`const`** в браузерах можно проверить по следующим адресам: **`http://kangax.github.io/compat-table/es6/#test-let`** и **`http://kangax.github.io/compat-table/es6/#test-const`**.

5.5.1. Изменяемость переменных

Если разделить упомянутые выше слова для объявления переменных по изменяемости, то ключевое слово `const` следует отложить в одну сторону, а ключевые слова `var` и `let` – в другую. С одной стороны, все переменные, объявляемые с помощью ключевого слова `const`, являются неизменяемыми, а следовательно, их значения устанавливаются только один раз. А с другой стороны, переменные, объявляемые с помощью ключевых слов `var` и `let`, являются обыкновенными переменными, значения которых можно изменять сколько угодно. А теперь выясним, каким образом действуют и ведут себя переменные типа `const`.

Переменные типа `const`

Переменная типа `const` подобна обычной переменной, за исключением того, что при объявлении следует сразу же присвоить ей инициализирующее значение, но в дальнейшем ей нельзя будет присваивать любое другое значе-

ние. Да, но какая же это переменная? Переменные типа `const` обычно применяются в двух совершенно разных целях.

- Указание переменных, которые не должны больше переназначаться (в примерах кода, представленных в остальной части данной книги, эти переменные употребляются именно с такой целью).
- Обращение к фиксированному значению, например, к максимальному числу ронинов в отряде (`MAX_RONIN_COUNT`) по имени вместо использования литерального значения вроде **234**. Благодаря этому упрощается понимание и сопровождение прикладных программ на JavaScript. Ведь в этом случае исходный код не испещрен произвольными, на первый взгляд, литералами (**234**), а содержит выразительные имена (`MAX_RONIN_COUNT`), значения которых указываются только один раз.

Но в любом случае прикладной код защищается от ненужных или случайных видоизменений, поскольку переменные типа `const` нельзя переназначать в ходе выполнения программы. А кроме того, интерпретатор JavaScript получает возможность оптимизировать производительность. В примере кода из листинга 5.6 наглядно демонстрируется поведение переменных типа `const`.

Листинг 5.6. Поведение переменных типа `const`

```
const firstConst = "samurai";
assert(firstConst === "samurai", "firstConst is a samurai");

try{
  firstConst = "ninja";
  fail("Shouldn't be here");
} catch(e){
  pass("An exception has occurred");
}

assert(firstConst === "samurai",
       "firstConst is still a samurai!");

const secondConst = {};

secondConst.weapon = "wakizashi";
assert(secondConst.weapon === "wakizashi",
       "We can add new properties");

const thirdConst = [];
assert(thirdConst.length === 0, "No items in our array");

thirdConst.push("Yoshi");

assert(thirdConst.length === 1, "The array has changed");
```

Определить переменную типа `const` и проверить присвоенное ей значение

При попытке присвоить новое значение переменной типа `const` генерируется исключение

Создать новую переменную типа `const` и присвоить ей новое значение

Переменной `secondConst` нельзя присвоить совершенно новый объект, но ничто не мешает видоизменить уже имеющийся объект

Это же относится и к массивам

В данном примере кода сначала определяется переменная `firstConst` типа `const` со значением `samurai`, а затем проверяется, была ли она инициализирована должным образом:

```
const firstConst = "samurai";
assert(firstConst === "samurai", "firstConst is a samurai");
```

Далее предпринимается следующая попытка присвоить совершенно новое значение `ninja` переменной `firstConst`:

```
try{
  firstConst = "ninja";
  fail("Shouldn't be here");
} catch(e){
  pass("An exception has occurred");
}
```

Переменная `firstConst`, по существу, является константой, и поэтому ей нельзя присвоить новое значение, а следовательно, интерпретатор JavaScript генерирует исключение, не изменяя значение этой переменной. Следует заметить, что в данном примере кода применяются два не употреблявшихся до сих пор метода: `fail()` и `pass()`. Эти методы ведут себя таким же образом, как и функция `assert()`, за исключением того, что проверка в методе `fail()` никогда не проходит, тогда как в методе `pass()` она проходит всегда. Эти методы применяются в данном примере кода с целью проверить, возникло ли исключение. Если исключение возникло, активизируется оператор `catch` и выполняется метод `pass()`. А в отсутствие исключения выполняется метод `fail()` и уведомляется, что дело обстоит не так, как должно быть. Таким образом, можно проверить, произошло ли исключение, как показано на рис. 5.10.

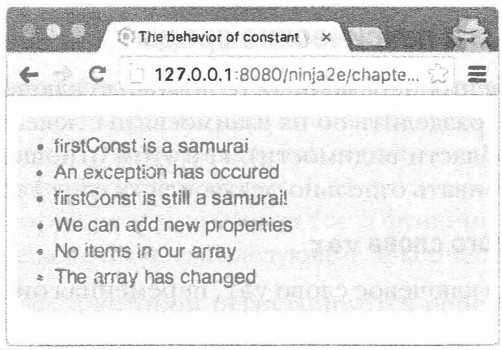


Рис. 5.10. Проверка поведения переменных типа **const**. Исключение возникает в том случае, если попытаться присвоить переменной типа **const** другое значение

Далее в рассматриваемом здесь примере кода определяется еще одна переменная типа `const`. На этот раз она инициализируется пустым объектом:

```
const secondConst = {};
```

Обсудим теперь особенности переменных типа const. Как было показано выше, переменным типа const нельзя присваивать другие значения. Но ничто не мешает *видоизменить* текущее значение. Например, в текущий объект можно ввести новые свойства, как показано ниже.

```
secondConst.weapon = "wakizashi";
assert(secondConst.weapon === "wakizashi",
       "We can add new properties");
```

А если переменная типа const ссылается на массив, то этот массив можно видоизменить до любой степени, как демонстрируется в следующем фрагменте кода:

```
const thirdConst = [];
assert(thirdConst.length === 0, "No items in our array");

thirdConst.push("Yoshi");

assert(thirdConst.length === 1, "The array has changed");
```

Вот, собственно, и все, что касается переменных типа const. Обращаться с ними крайне просто. Достаточно запомнить, что значение переменной типа const может быть установлено только при ее инициализации и что ей нельзя в дальнейшем присваивать другие значения. Но в то же время можно видоизменить существующее значение, хотя и полностью его переопределить нельзя.

Итак, исследовав изменяемость переменных, перейдем к подробному рассмотрению взаимосвязей между различными типами переменных и лексическими средами.

5.5.2. Ключевые слова для определения переменных и лексические среды

Три вида определения переменных (с помощью ключевых слов var, let и const) можно также разделить по их взаимосвязи с лексической средой (иными словами, по их области видимости). И в этом отношении ключевое слово var следует рассматривать отдельно от ключевых слов let и const.

Применение ключевого слова var

Если применяется ключевое слово var, переменная определяется в ближайшей лексической среде контекста функции или глобального контекста. (Однако блоки при этом игнорируются!) Эта давнишняя особенность языка JavaScript служила помехой для многих разработчиков, переходящих на JavaScript с других языков программирования. В качестве примера рассмотрим код из листинга 5.7.

Листинг 5.7. Применение ключевого слова var

```
var globalNinja = "Yoshi";
```

Определить глобальную переменную с помощью ключевого слова var

```
function reportActivity(){
```

```
  var functionActivity = "jumping";

  for(var i = 1; i < 3; i++) {
    var forMessage = globalNinja + " " + functionActivity;
    assert(forMessage === "Yoshi jumping",
           "Yoshi is jumping within the for block");
    assert(i, "Current loop counter:" + i);
  }

  assert(i === 3 && forMessage === "Yoshi jumping",
         "Loop variables accessible outside of the loop");
}

reportActivity();

assert(typeof functionActivity === "undefined"
       && typeof i === "undefined" && typeof forMessage === "undefined",
       "We cannot see function variables outside of a function");
```

Определить локальную переменную в функции с помощью ключевого слова `var`

Определить две переменные в цикле `for` с помощью ключевого слова `var`

В цикле `for` доступны переменные из блока, функции и глобального кода, что не удивительно

Но переменные, употребляемые в цикле `for`, доступны и за его пределами

Как правило, ни одна из переменных функции недоступна за ее пределами

Сначала в данном примере кода определяется глобальная переменная `globalNinja`, а затем функция `reportActivity()`, где в течение двух итераций цикла уведомляется о прыгающих действиях ниндзя, имя которого задано в переменной `globalNinja`. Как видите, в теле цикла `for` вполне доступны переменные из блока (`i` и `forMessage`) и функции (`functionActivity`), а также глобальные переменные (`globalNinja`).

Но многих разработчиков, перешедших на JavaScript из других языков программирования, удивляет, что переменные, определенные в блоках кода, могут быть доступны даже за пределами этих блоков, как показано ниже.

```
assert(i === 3 && forMessage === "Yoshi jumping",
       "Loop variables accessible outside of the loop");
```

Эта особенность объясняется тем, что переменные, объявляемые с помощью ключевого слова `var`, всегда регистрируются в ближайшей лексической среде (функции или глобальной), не обращая никакого внимания на блоки. Такая ситуация наглядно демонстрируется на рис. 5.11, где показано состояние лексических сред после второй итерации цикла `for` в функции `reportActivity()`.

В данном примере кода имеются следующие лексические среды.

- Глобальная среда, в которой регистрируется переменная `globalNinja`, поскольку это ближайшая лексическая среда (функции или глобальная).
- Среда функции `reportActivity()`, создаваемая при вызове данной функции и содержащая переменные `functionActivity`, `i` и `forMessage`, поскольку они определяются с помощью ключевого слова `var`, и она оказывается средой ближайшей функции.
- Среда блока цикла `for`, которая оказывается пустой, поскольку переменные, определяемые с помощью ключевого слова `var`, игнорируют блоки кода, даже если они в них содержатся.

Вследствие столь необычного поведения в стандарте ES6 языка JavaScript были внедрены два новых ключевых слова, `let` и `const`, для определения переменных.

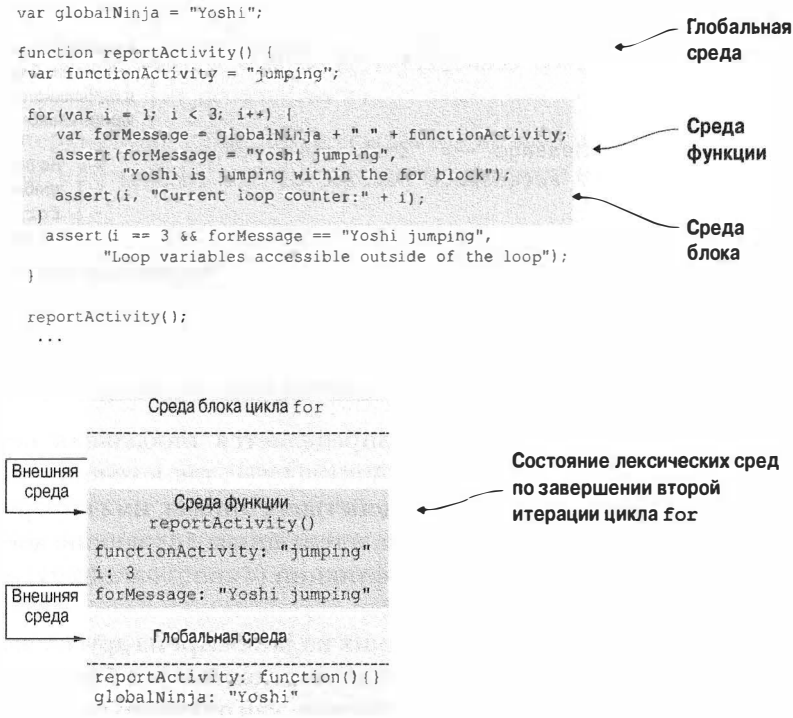


Рис. 5.11. С помощью ключевого слова **`var`** переменные определяются в ближайшей среде функции или в глобальной среде, пренебрегая средами блоков. В данном случае переменные **`forMessage`** и **`i`** регистрируются в среде функции **`reportActivity()`**, т.е. в ближайшей лексической среде функции, несмотря на то, что они содержатся в цикле **`for`**

Обозначение переменных в области видимости блоков с помощью ключевых слов `let` и `const`

В отличие от ключевого слова `var`, определяющего переменную в ближайшей лексической среде (функции или глобальной), ключевые слова `let` и `const` действуют проще. Они определяют переменные в ближайшей лексической среде, которая может быть средой блока кода, цикла, функции или даже глобальной средой. С помощью ключевых слов `let` и `const` можно определить переменные в области видимости блока, функции или глобального кода. В листинге 5.8 приведен исходный код из предыдущего примера, переписанный с целью продемонстрировать применение ключевых слов `let` и `const`.

Листинг 5.8. Применение ключевых слов `let` и `const`

```
const GLOBAL_NINJA = "Yoshi";

function reportActivity(){
  const functionActivity = "jumping";

  for(let i = 1; i < 3; i++) {
    let forMessage = GLOBAL_NINJA + " " + functionActivity;
    assert(forMessage === "Yoshi jumping",
           "Yoshi is jumping within the for block");
    assert(i, "Current loop counter:" + i);
  }

  assert(typeof i === "undefined" && typeof forMessage === "undefined",
         "Loop variables not accessible outside the loop");
}

reportActivity();
assert(typeof functionActivity === "undefined"
       && typeof i === "undefined"
       && typeof forMessage === "undefined",
       "We cannot see function variables outside of a function");
```

Определить глобальную переменную с помощью ключевого слова const. Глобальные переменные типа const обычно обозначают прописными буквами

Определить локальную переменную в функции с помощью ключевого слова const

Определить две переменные в цикле for с помощью ключевого слова let

В цикле for доступны переменные из блока, функции и глобального кода, что не удивительно

Теперь переменные из цикла for недоступны за его переделами

Как правило, ни одна из переменных функции недоступна за ее пределами

На рис. 5.12 наглядно показано текущее состояние лексических сред из данного примера кода, когда завершается вторая итерация цикла `for` в теле функции `reportActivity()`. И в этом случае имеются три лексические среды: глобальная среда (для выполнения глобального кода за пределами всех функций и блоков), среда функции `reportActivity()`, привязанная к данной функции, а также среда блока для выполнения кода в теле цикла `for`. Но поскольку в данном примере кода применяются ключевые слова `let` и `const`, то переменные определяются в ближайшей лексической среде. В частности, переменная `GLOBAL_NINJA` определяется в глобальной среде, переменная `functionActivity` — в среде функции `reportActivity()`, а переменные `i` и `forMessage` — в среде блока цикла `for`.

Теперь, когда в стандарт наконец-то введены ключевые слова `const` и `let`, множество новых разработчиков, которые лишь недавно перешли на JavaScript из других языков программирования, могут быть вполне удовлетворены. Наконец-то в JavaScript поддерживаются такие же правила соблюдения областей видимости, как и в других C-подобных языках. Именно поэтому в примерах кода, приведенных далее в книге, практически всегда употребляются ключевые слова `const` и `let` вместо ключевого слова `var`.

```
const GLOBAL_NINJA = "Yoshi";

function reportActivity() {
 const functionActivity = "jumping";

 for(let i = 1; i < 3; i++) {
    let forMessage = GLOBAL_NINJA + " " + functionActivity;
    assert(forMessage = "Yoshi jumping",
           "Yoshi is jumping within the for block");
    assert(i, "Current loop counter:" + i);
 }
  assert(typeof i == "undefined"
      && typeof forMessage == "undefined",
        "Loop variables not accessible outside the loop");
}

reportActivity();
...
```

Глобальная среда

Среда функции

Среда блока

Рис. 5.12. С помощью ключевых слов `let` и `const` переменные определяются в ближайшей среде. В данном случае переменные **`forMessage`** и `i` регистрируются в среде блока цикла **`for`**, переменная **`functionActivity`** — в среде функции **`reportActivity()`**, а переменная **`GLOBAL_NINJA`** — в глобальной среде. И в каждом случае эти лексические среды являются ближайшими для соответствующих переменных

5.5.3. Регистрация идентификаторов в лексических средах

Один из движущих принципов, положенных в основу разработки JavaScript как языка программирования, состоял в простоте его применения. И в этом кроется одна из главных причин, по которым не указываются типы значений, возвращаемых из функций, параметров функций, переменных и т.д. И как вам должно быть уже известно, код JavaScript выполняется построчно и непосредственно. Рассмотрим в качестве примера следующий фрагмент кода:

```
firstRonin = "Kiyokawa";
secondRonin = "Kondo";
```

Сначала идентификатору `firstRonin` присваивается значение `"Kiyokawa"`, а затем идентификатору `secondRonin` — значение `"Kondo"`. И в этом нет ничего удивительного, не так ли? Тем не менее рассмотрим следующий пример:

```
const firstRonin = "Kiyokawa";
check(firstRonin);
function check(ronin) {
  assert(ronin === "Kiyokawa", "The ronin was checked! ");
}
```

В данном случае значение `"Kiyokawa"` сначала присваивается идентификатору, а затем вызывается функция `check()` с идентификатором `firstRonin` в качестве параметра. Но позвольте, если код выполняется построчно, то можно ли вообще вызывать функцию `check()`? Ведь интерпретатор еще не достиг объявления этой функции, и поэтому ему ничего о ней неизвестно.

Но если протестировать приведенный выше фрагмент кода, как показано на рис. 5.13, то обнаружится, что он вполне работоспособен. Язык JavaScript не особенно придирчив к месту определения функций. В частности, объявления функций можно размещать как до, так и после их вызовов. И это не должно особенно беспокоить разработчиков.

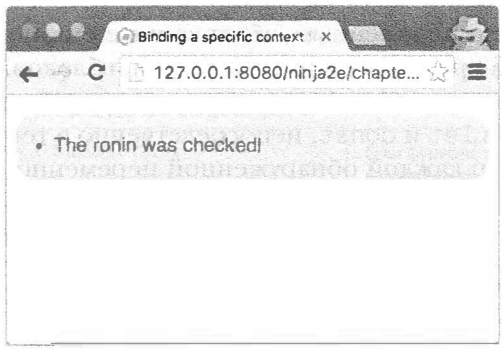


Рис. 5.13. Функция в действительности доступна — даже до ее фактического определения

Процесс регистрации идентификаторов

Но если отложить в сторону простоту применения, то каким образом интерпретатору JavaScript удалось выяснить, что функция `check()` существует, если код выполняется построчно? Оказывается, что интерпретатор JavaScript немного "хитрит", и на самом деле выполнение кода JavaScript происходит в два этапа.

Первый этап активизируется всякий раз, когда создается новая лексическая среда. На данном этапе код еще не выполняется, а интерпретатор JavaScript просматривает и регистрирует все переменные и функции, объявленные в текущей лексической среде. После этого начинается второй этап выполнения

кода JavaScript. Конкретное поведение зависит от типа переменной (`let`, `var`, `const`, объявления функции) и типа среды (глобальной, функции или блока).

Данный процесс происходит следующим образом.

1. Если создается среда функции, то создается идентификатор неявного параметра `arguments` наряду со всеми формальными параметрами функции и значениями их аргументов. А если это не среда функции, то данный шаг пропускается.
2. Если создается глобальная среда или же среда функции, то в текущем коде, не доходя до тела других функций, просматриваются объявления функций, но не функциональные выражения или стрелочные функции! Для каждого обнаруженного объявления функции создается новая функция, которая привязывается к идентификатору в среде по своему имени. Если этот идентификатор уже существует, его значение перезаписывается. А если это одна из сред блоков, то данный шаг пропускается.
3. Текущий код проверяется на наличие объявлений переменных. В средах функций и глобальных функциях обнаруживаются все переменные, объявленные с помощью ключевого слова `var` и определенные за пределами других функций, хотя их можно разместить в блоках, а также обнаруживаются все переменные, объявленные с помощью ключевых слов `let` и `const` за пределами других функций и блоков. В средах блоков код проверяется только на наличие переменных, объявленных с помощью ключевых слов `let` и `const`, непосредственно в текущем блоке. Если же идентификатор каждой обнаруженной переменной не существует в текущей среде, он регистрируется и инициализируется неопределенным значением (`undefined`). Но если идентификатор существует, то он оставляется со своим значением.

Все эти шаги подытожены на рис. 5.14.

А теперь рассмотрим следствия из упомянутых выше правил. Соблюдение этих правил при написании кода JavaScript способно вызвать ряд характерных трудностей, которые могут привести к едва уловимым и не совсем понятным программным ошибкам. Начнем с выяснения причин, по которым функцию можно вызывать еще до ее объявления.

Вызов функций до их объявления

Одна из примечательных особенностей языка JavaScript состоит в том, что порядок определения функций не имеет особого значения. Те, кто некогда программировал на Pascal, возможно, уже не помнят его жестко структурированные требования. В языке JavaScript допускается вызывать функцию еще до ее формального объявления, как демонстрируется в примере кода из листинга 5.9.

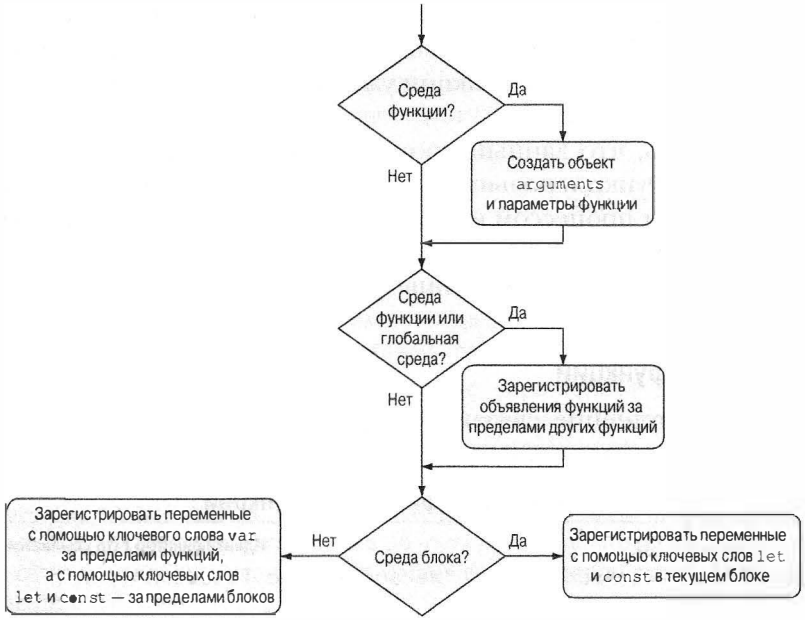


Рис. 5.14. Процесс регистрации идентификаторов в зависимости от типа среды

Листинг 5.9. Доступ к функции до ее объявления

```
assert(typeof fun === "function",
      "fun is a function even though its
       definition isn't reached yet!");

assert(typeof myFunExp === "undefined",
       "But we cannot access function expressions");

assert(typeof myArrow === "undefined",
       "Nor arrow functions");

function fun(){}

var myFunExpr = function(){};
var myArrow = (x) => x;
```

Если функция определяется с помощью оператора объявления функции, она может быть доступна до момента своего определения

Функции, определяемые в виде функциональных выражений или стрелочных функций, заранее недоступны

Функция `fun()` определяется в виде оператора объявления функции

Переменная `myFunExpr` указывает на функциональное выражение, а переменная `myArrow` — на стрелочную функцию

Функция `fun()` может быть доступна еще до момента своего определения. И это вполне допустимо потому, что функция `fun()` определяется в виде оператора объявления функции. Как пояснялось на втором шаге рассмотренного ранее процесса, если функции создаются с помощью операторов объявлений функций, их идентификаторы регистрируются при создании текущей лексической среды, т.е. *еще до* выполнения любого кода JavaScript. Следовательно, прежде чем начнется вызов функции `assert()`, функция `fun()` уже существует.

Интерпретатор JavaScript упрощает задачу разработчиков, позволяя им заранее обращаться к функциям, не особенно заботясь о конкретном порядке расположения функций. Ведь функции уже существуют к тому моменту, когда начинает выполняться код.

Следует заметить, что данный процесс распространяется только на объявления функций. Функциональные выражения и стрелочные функции не затрагиваются данным процессом и создаются в тот момент, когда выполнение программы достигает их определений. Именно поэтому в рассматриваемом здесь примере кода заранее недоступны функции, на которые указывают переменные `myFunExp` и `myArrow`.

Переопределение функций

Следующее затруднение связано с переопределением идентификаторов функций. Рассмотрим еще один пример кода, приведенный в листинге 5.10.

Листинг 5.10. Переопределение идентификаторов функций

```
assert(typeof fun === "function", "We access the function");

var fun = 3;

assert(typeof fun === "number", "Now we access the number");

function fun(){}

assert(typeof fun === "number", "Still a number");
```

Идентификатор `fun` ссылается на функцию

Определить переменную `fun` и присвоить ей число

Идентификатор `fun` ссылается на число

Объявить функцию `fun()`

Идентификатор `fun` по-прежнему ссылается на число

В данном примере кода переменная и функция объявляются с одинаковым именем `fun`. Если выполнить этот код, то окажется, то что все утверждения в нем проходят. В первом утверждении идентификатор `fun` ссылается на функцию, а во втором и третьем утверждении — на число.

Такое поведение является прямым следствием шагов, предпринятых при регистрации идентификаторов. На втором шаге описанного ранее процесса функции, определяемые с помощью оператора объявления функции, создаются и связываются со своими идентификаторами перед интерпретацией любого кода, на третьем шаге обрабатываются объявления переменных, а неопределенное значение (`undefined`) связывается с теми идентификаторами, которые еще не встречались в текущей среде.

В данном случае значение `undefined` не присваивается переменной `fun`, поскольку идентификатор `fun` уже встречался на втором шаге рассматриваемого здесь процесса, когда регистрировались объявления функций. Именно поэтому проходит первое утверждение, в котором проверяется, соответствует ли идентификатор `fun` функции. После этого следует оператор присваивания, `var fun = 3`, где число **3** присваивается переменной, обозначаемой идентифи-

катором `fun`. Вследствие этого теряется ссылка на функцию, и далее идентификатор `fun` ссылается на число.

В ходе выполнения конкретной программы объявления функций пропускаются. Таким образом, определение функции `fun()` не оказывает никакого влияния на значение идентификатора `fun`.

Поднятие переменных

При пояснении механизма поиска идентификаторов в литературе и блогах по JavaScript нередко можно встретить термин *поднятие* (*hoisting*). Например, объявления переменных и функций поднимаются до верхнего края области видимости функции или глобальной области видимости.

Но, как было показано выше, это довольно упрощенное представление. Объявления переменных и функций технически никуда не "перемещаются". Они просматриваются и регистрируются в лексических средах еще до выполнения любого кода. И хотя *поднятие* дает основное представление о соблюдении областей видимости в JavaScript, тем не менее, рассмотрев лексические среды, мы дали более углубленное представление об этом процессе, чтобы вы смогли продвинуться дальше на пути становления настоящего мастера программирования на JavaScript.

В следующем разделе все понятия, рассмотренные до сих пор в этой главе, помогут вам лучше понять замыкания.

5.6. Исследование принципа действия замыканий

В начале этой главы были рассмотрены замыкания — механизм, предоставляющий функции доступ ко всем переменным, которые оказываются в области видимости при создании самой функции. Помимо этого, было показано, каким образом замыкания могут помочь в написании кода на JavaScript, позволяя, например, сымитировать частные объектные переменные или сделать код более изящным, когда приходится иметь дело с обратными вызовами.

Замыкания тесно и бесповоротно связаны с областями видимости. Замыкания являются побочным эффектом, непосредственно вытекающим из правил соблюдения области видимости, принятых в JavaScript. Поэтому мы еще раз вернемся в этом разделе к примерам замыканий, представленным ранее в главе. Но на этот раз мы выгодно воспользуемся контекстами выполнения и лексическими средами, чтобы лучше уяснить внутренний механизм действия замыканий.

5.6.1. Еще раз об имитации закрытых переменных с помощью замыканий

Как пояснялось ранее, замыкания могут оказывать помощь в имитации закрытых переменных. Итак, разобравшись в правилах соблюдения области ви-

димости, принятых в JavaScript, вернемся к рассмотренным ранее примерам закрытых переменных, уделив на этот раз основное внимание контекстам выполнения и лексическим средам. Чтобы упростить дело, приведем еще раз пример кода из листинга 5.3, как показано в листинге 5.11.

Листинг 5.11. Приближенное представление закрытых переменных с помощью замыканий

Вызвать метод `feint()`, в котором наращивается подсчет количества ложных выпадов со стороны ниндзя

Объявить переменную в функции-конструкторе. Область видимости этой переменной ограничивается телом конструктора, поэтому она оказывается закрытой переменной

Метод для инкрементирования значения переменной `feints`. Это закрытая переменная, и поэтому ее значение никто не может изменить без разрешения. Посторонним предоставляется ограниченный доступ к ней через методы

Проверить, выполнено ли приращение

Проверить, можно ли получить прямой доступ к переменной

Когда новый объект `ninja2` создается с помощью конструктора объектов типа `Ninja`, ему назначается новая копия переменной `feints`

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}

var ninja1 = new Ninja();
ninja1.feint();

assert(ninja1.feints === undefined,
       "And the private data is inaccessible to us.");
assert(ninja1.getFeints()=== 1,
       "We're able to access the internal feint count.");

var ninja2 = new Ninja();
assert(ninja2.getFeints()=== 0,
       "The second ninja object gets its own feints variable.");
```

А теперь проанализируем состояние приведенного выше прикладного кода после создания первого объекта типа `Ninja` (рис. 5.15). Зная теперь внутренний механизм поиска идентификаторов, можно лучше понять, каким образом замыкания вступают в действие в данной ситуации. Конструкторы в JavaScript являются функциями, вызываемыми с ключевым словом `new`. Следовательно, *всякий* раз, когда вызывается функция-конструктор, создается *новая* лексическая среда, где располагаются переменные, являющиеся локальными для конструктора. В данном примере создается новая среда конструктора объектов типа `Ninja`, где и располагается переменная `feints`.

Кроме того, всякий раз, когда создается функция, в ней сохраняется ссылка на лексическую среду, в которой она была создана (через внутреннее свойство `[[Environment]]`). В данном случае в конструкторе объектов типа `Ninja` создаются две новые функции, `getFeints()` и `feint()`, которые ссылаются на среду конструктора объектов типа `Ninja`, поскольку именно в этой среде они и были созданы.

Рис. 5.15. Закрытые переменные реализуются в виде замыканий, которые создаются методами объектов, определяемыми в конструкторе

Функции `getFeints()` и `feint()` присваиваются методам вновь созданного объекта, который, как упоминалось в предыдущей главе, доступен по ссылке, обозначаемой ключевым словом `this`. Следовательно, обе эти функции доступны за пределами конструктора объектов типа `Ninja`, что, в свою очередь, позволяет сделать вывод, что вокруг переменной `feints`, по существу, образовалось замыкание.

Когда создается еще один объект типа `Ninja` (т.е. объект `ninja2`), весь описанный выше процесс повторяется. На рис. 5.16 наглядно показано состояние рассматриваемого здесь прикладного кода после создания второго объекта типа `Ninja`.

Каждый объект, создаваемый с помощью конструктора `Ninja`, получает свои методы, которые охватывают переменные, определенные при вызове этого конструктора, причем метод `ninja1.getFeints()` отличается от метода `ninja2.getFeints()`. Эти "закрытые переменные" доступны не напрямую, а только через методы объектов, создаваемых в конструкторе!

А теперь выясним, что происходит при вызове метода `ninja2.getFeints()`. Подробности этого вызова наглядно показаны на рис. 5.17.

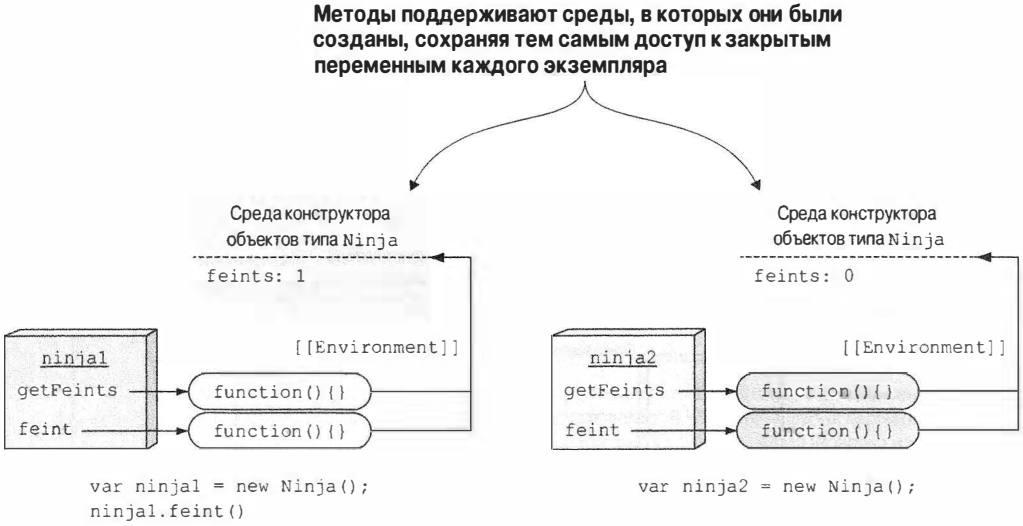


Рис. 5.16. Методы каждого экземпляра образуют замыкания вокруг закрытых переменных экземпляра

Прежде чем делать вызов `ninja2.getFeints()`, интерпретатор JavaScript выполняет глобальный код. Выполнение рассматриваемого здесь прикладного кода происходит в глобальном контексте выполнения, т.е. в единственном контексте, находящемся в стеке контекстов выполнения. В то же время единственной активной лексической средой является глобальная среда, связанная непосредственно с глобальным контекстом выполнения.

При выполнении вызова `ninja2.getFeints()` вызывается метод `getFeints()` объекта `ninja2`. А поскольку вызов каждой функции приводит к созданию нового контекста выполнения, то новый контекст выполнения функции `getFeints()` создается и размещается в стеке контекстов выполнения. Это также приводит к созданию новой лексической среды функции `getFeints()`, которая обычно используется для размещения переменных, определенных в данном методе. Кроме того, лексическая среда функции `getFeints()` получает как внешнюю ту среду, в которой была создана функция `getFeints()`, т.е. среду конструктора объектов типа `Ninja`, которая была активна при создании объекта `ninja2`.

Выясним далее, что произойдет при попытке получить значение переменной `feints`. Прежде всего, происходит обращение к активной в настоящий момент лексической среде функции `getFeints()`. А поскольку в функции `getFeints()` не определяются никакие переменные, то данная лексическая среда пуста и найти в ней искомую переменную `feints` не удастся. Далее поиск продолжается в среде, *внешней* по отношению к текущей лексической среде (в данном случае в среде конструктора объектов типа `Ninja`, которая была активна при создании объекта `ninja2`). На сей раз в среде конструктора объек-

тов типа Ninja содержится ссылка на переменную feints, и на этом поиск завершается.

Теперь, когда стала понятна роль контекстов выполнения и лексических сред при обращении с замыканиями, уделим внимание "закрытым" переменным и выясним, почему их название заключено в кавычки. Как вам должно быть уже понятно, эти "закрытые" переменные на самом деле не являются закрытыми свойствами объекта, а переменными, с которыми работают методы объектов, создаваемых в конструкторе. Рассмотрим далее один интересный побочный эффект, вытекающий из этого.

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}

var ninja1 = new Ninja();
var ninja1.feint();

var ninja2 = new Ninja();
ninja2.getFeints();
```

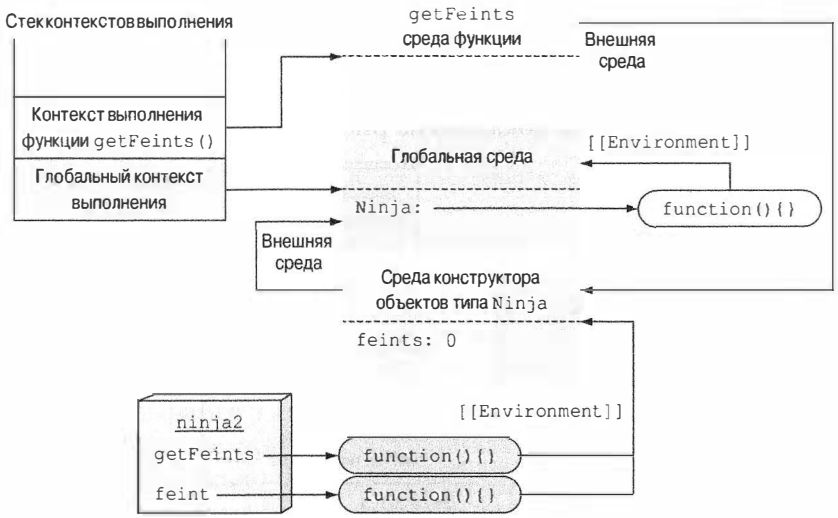


Рис. 5.17. Состояние контекстов выполнения и лексических сред при выполнении вызова **ninja2.getFeints()**. Создается новая среда функции **getFeints()**, для которой среда функции-конструктора, где был создан объект **ninja2**, оказывается внешней. В функции **getFeints()** может быть доступна закрытая переменная **feints**

5.6.2. Разъяснение по поводу закрытых переменных

Программирующим на JavaScript ничто не мешает присваивать свойства, созданные в одном объекте, другому объекту. Например, исходный код из листинга 5.11 можно без труда переписать так, как показано в листинге 5.12.

Листинг 5.12. Закрытые переменные доступны через функции, а не объекты!

```
function Ninja() {
  var feints = 0;
  this.getFeints = function(){
    return feints;
  };
  this.feint = function(){
    feints++;
  };
}
var ninja1 = new Ninja();
ninja1.feint();

var imposter = {};
imposter.getFeints = ninja1.getFeints;

assert(imposter.getFeints() === 1,
       "The imposter has access to the feints variable!");
```

Сделать метод `getFeints()` объекта `ninja1` доступным через объект `imposter`

Проверить, доступна ли предположительно закрытая переменная объекта `ninja1`

Исходный код из листинга 5.12 видоизменен таким образом, чтобы присвоить метод `ninja1.getFeints()` совершенно новому объекту `imposter`. После этого метод `getFeints()` можно вызвать для объекта `imposter` и проверить, доступно ли значение переменной `feints`, созданной при получении экземпляра объекта `ninja1`. Тем самым доказывается, что закрытая переменная на самом деле имитируется, как показано на рис. 5.18.

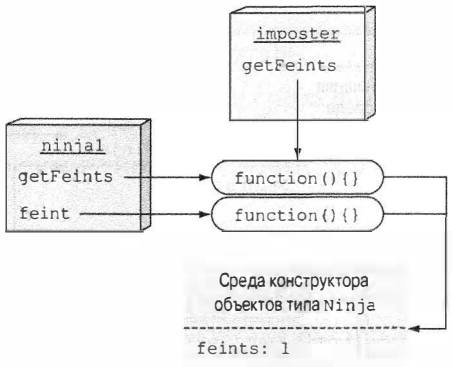


Рис. 5.18. Закрытые переменные доступны через функции, даже если функция принадлежит другому объекту!

Рассматриваемый здесь пример наглядно показывает, что в JavaScript вообще отсутствуют закрытые объектные переменные, но для "достаточно хорошей" их имитации можно воспользоваться замыканиями, образуемыми методами объектов. И хотя это не настоящие закрытые переменные, многие разработчики считают, что они удобны для сокрытия информации.

5.6.3. Еще раз о замыканиях и обратных вызовах

Вернемся к рассмотренному ранее простому примеру анимации с помощью таймеров обратного вызова. На этот раз анимируем два объекта, как показано в примере кода из листинга 5.13.

Листинг 5.13. Применение замыкания при обратном вызове через определенный промежуток времени срабатывания таймера

```
<div id="box1">First Box</div>
<div id="box2">Second Box</div>
<script>
  function animateIt(elementId) {
    var elem = document.getElementById(elementId);
    var tick = 0;
    var timer = setInterval(function(){
      if (tick < 100) {
        elem.style.left = elem.style.top = tick + "px";
        tick++;
      }
      else {
        clearInterval(timer);
        assert(tick === 100,
               "Tick accessed via a closure.");
        assert(elem,
               "Element also accessed via a closure.");
        assert(timer,
               "Timer reference also obtained via a closure." );
      }
    }, 10);
  }
  animateIt("box1");
  animateIt("box2");
</script>
```

Как пояснялось ранее в этой главе, с помощью замыканий упрощается анимация нескольких объектов на веб-страницах. Но на этот раз мы рассмотрим для этой цели лексические среды, как показано на рис. 5.19.

Всякий раз, когда вызывается функция `animateIt()`, для нее создается новая лексическая среда ❶ ❷, где отслеживаются значения ряда переменных, от которых зависит анимация (`elementId`, `elem` — элемент, подвергаемый анимации; `tick` — текущее число тактов анимации; `timer` — идентификатор таймера, выполняющего анимацию). Эта среда поддерживается активной до тех пор,

```
<div id="box1">First box</div>
<div id="box2">Second box</div>
<script>
function animateIt(elementId) {
   var elem = document.getElementById(elementId);
   var tick = 0;
    var timer = setInterval(function (){
     if (tick < 100) {
       var position = tick + "px";
       elem.style.left = position;
       elem.style.top = position;
       tick++;
     }
     else {
      clearInterval(timer);
     }
   }, 10);
}
animateIt("box1");
animateIt("box2");
</script>
```

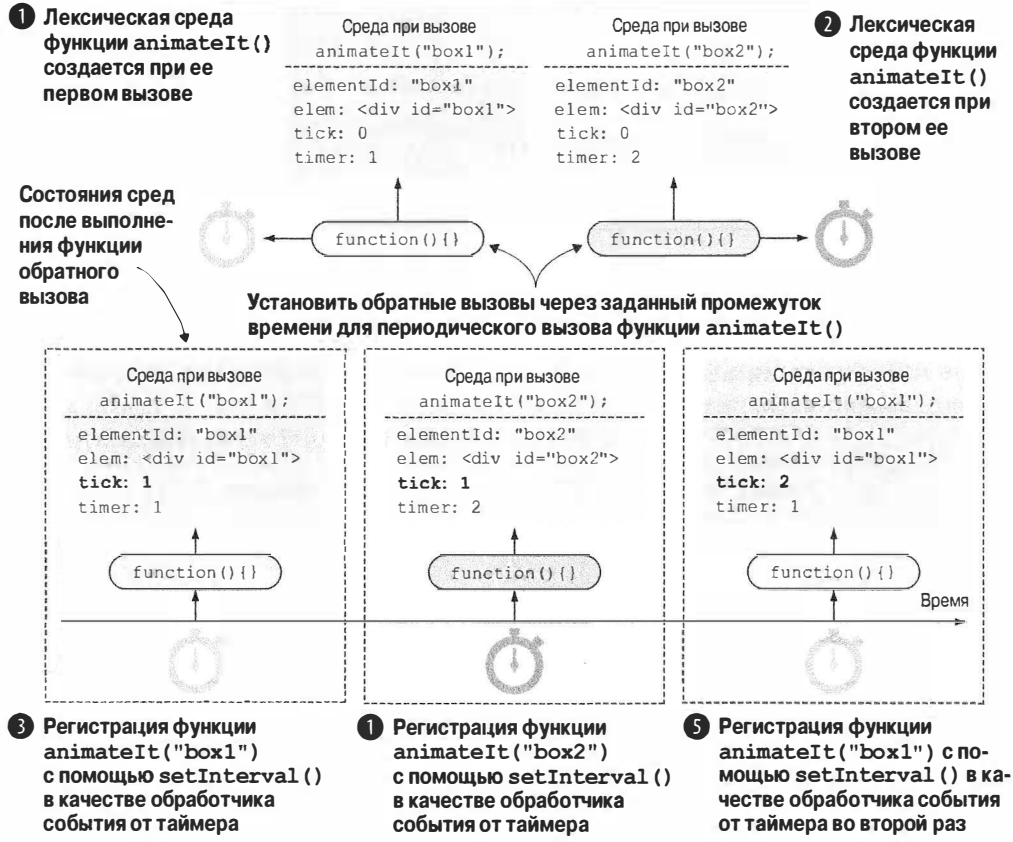


Рис. 5.19. Образуя несколько замыканий, можно выполнять одновременно несколько действий. Всякий раз, когда истекает заданный промежуток времени, функция обратного вызова повторно активизирует среду, которая была активна в момент создания обратного вызова. В замыкании каждой функции обратного вызова автоматически отслеживаются значения ее собственных переменных

пока хотя бы одна функция обращается к ее переменным через замыкания. В данном случае браузер будет поддерживать активной среду для функции обратного вызова `setInterval()` до тех пор, пока не будет вызвана функция `clearInterval()`. А когда заданный промежуток времени исчерпывается, браузер делает соответствующий обратный вызов, и вместе с ним становятся доступными (через замыкания) переменные, определенные при создании функции обратного вызова. Благодаря этому удается избежать хлопот, связанных с преобразованием обратного вызова и активных переменных вручную ❸ ❹ ❺, значительно упростив тем самым исходный код.

Вот, собственно, и все, что следовало бы сказать о замыканиях и областях видимости. Подведя краткие итоги этой главы и предложив упражнения для закрепления ее материала, перейдем к рассмотрению в следующей главе совершенно новых понятий генераторов и обещаний, введенных в стандарт ES6 с целью оказывать помощь разработчикам в написании асинхронного кода.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Замыкания предоставляют функции доступ ко всем переменным, находившимся в области видимости в момент определения самой функции. Они образуют "защитную оболочку" вокруг функции и переменных, находившихся в области видимости в момент ее определения. Благодаря этому функция получает в свое распоряжение все, что ей требуется для выполнения, даже если область видимости, в которой функция была создана, давно канула в Лету.
- Замыканиями можно воспользоваться для реализации следующих средств.
  - Имитации закрытых объектных переменных путем охвата переменных конструктора через замыкания методов.
  - Применение обратных вызовов, значительно упрощающих прикладной код.
- Интерпретатор JavaScript отслеживает выполнение функции через стек контекстов выполнения (или стек вызовов). Всякий раз, когда вызывается функция, новый контекст ее выполнения создается и размещается в стеке. Как только выполнение функции завершается, соответствующий контекст выполнения удаляется из стека.
- Интерпретатор JavaScript отслеживает идентификаторы с лексическими средами, иначе называемыми областями видимости.
- В языке JavaScript можно определить переменные с областью видимости глобального кода, функции и даже блока кода.
- Для определения переменных служат ключевые слова `var`, `let` и `const`. В частности:

  - ключевое слово `var` определяет переменную в ближайшей области видимости функции или глобальной области видимости, пренебрегая блоками кода;
  - ключевые слова `let` и `const` определяют переменную в ближайшей области видимости, включая блоки кода, позволяя создавать переменные с областью видимости блока, что было невозможно до принятия стандарта ES6 языка JavaScript. Кроме того, ключевое слово `const` позволяет определить "переменные", значения которым присваиваются только один раз.

- Замыкания являются лишь побочным эффектом, вытекающим из правил соблюдения области видимости в JavaScript. Функцию можно вызывать даже в том случае, если область видимости, в которой она была создана, давно канула в Лету.

Упражнения

1. Замыкания предоставляют функциям доступ к внешним переменным, находившимся в области видимости:
   - а) при определении функции;
   - б) при вызове функции.
2. Замыкания требуют затрат:
   - а) кода;
   - б) оперативной памяти;
   - в) на обработку.
3. Отметьте в следующем примере кода идентификаторы, доступные через замыкания:

```
function Samurai(name) {
  var weapon = "katana";
  this.getWeapon = function(){
    return weapon;
  };

  this.getName = function(){
    return name;
  }

  this.message = name + " wielding a " + weapon;

  this.getMessage = function(){
    return this.message;
  }
}

var samurai = new Samurai("Hattori");
```

```
samurai.getWeapon();
samurai.getName();
samurai.getMessage();
```

4. Сколько контекстов выполнения создается в приведенном ниже фрагменте кода и каков наибольший размер стека контекстов выполнения?

```
function perform(ninja) {
  sneak(ninja);
  infiltrate(ninja);
}

function sneak(ninja) {
  return ninja + " skulking";
}

function infiltrate(ninja) {
  return ninja + " infiltrating";
}
perform("Kuma");
```

5. Какое ключевое слово JavaScript позволяет определить переменные, которым нельзя повторно присвоить другое значение?
6. В чем отличие ключевых слов `var` и `let`?
7. Где и почему генерируется исключение в приведенном ниже фрагменте кода?

```
getNinja();
getSamurai();

function getNinja() {
  return "Yoshi";
}

var getSamurai = () => "Hattori";
```

6 Функции на перспективу: генераторы и обещания

В этой главе...

- Продолжение выполнения функций с помощью генераторов
- Обработка асинхронных задач с помощью обещаний
- Достижение изящности асинхронного кода с помощью генераторов и обещаний

В трех предыдущих главах основное внимание было уделено функциям и, в частности, их определению и применению с наибольшей эффективностью. И хотя в этих главах были представлены некоторые из новых языковых средств, появившихся в стандарте ES6, в том числе стрелочные функции и области видимости блоков кода, в них все же исследовались те языковые средства, которые уже давно существуют в JavaScript. А в этой главе представлены *генераторы* и *обещания* — два новейших языковых средства, введенных в стандарт ES6.

Примечание

Как упоминалось выше, генераторы и обещания введены в стандарт ES6 языка JavaScript. Их текущую поддержку можно проверить по следующим адресам: `http://kangax.github.io/compat-table/es6/#test-generators` и `http://kangax.github.io/compat-table/es6/#test-Promise`.

Генераторы являются функциями особого типа. Если стандартная функция вычисляет и возвращает не больше одного значения при выполнении ее кода от начала и до конца, то генераторы могут вычислять и возвращать подряд не-

сколько значений (по одному для каждого запроса), приостанавливая свое выполнение между подобными запросами. И хотя генераторы являются новыми языковыми средствами в JavaScript, они давно уже существуют в таких языках, как Python, PHP и C#.

С одной стороны, генераторы нередко считаются наиболее непривычным и даже радикальным языковым средством, нечасто применяемым программистами. И хотя бо́льшая часть примеров, приведенных в этой главе, служит для наглядной демонстрации принципа действия функций-генераторов, мы исследуем некоторые практические вопросы применения генераторов. В ней будет также показано, как пользоваться генераторами с целью упростить изощренные циклы, а также приостановить и возобновить действие генераторов, что может оказать помощь в написании более простого и изящного асинхронного кода.

А с другой стороны, *обещания*[1] *(promises)* являются новым встроенным типом объектов, помогающим в работе с асинхронным кодом. Обещание – это заполнитель значения, которого еще нет, но оно ожидается в какой-то последующий момент. Обещания особенно удобны для работы в несколько асинхронных стадий.

В этой главе будет показано, каким образом действуют генераторы и обещания, а в завершение главы поясняется, как оптимально сочетать их вместе, чтобы упростить работу с асинхронным кодом. Но прежде чем вдаваться в подробности, рассмотрим вкратце, насколько изящным может быть асинхронный код.

Знаете ли вы?

Где применяется чаще всего функция-генератор?

Почему обещания более пригодны для асинхронного кода, чем обычные обратные вызовы?

Когда обещание выполняется и когда оно отклоняется, если вызвать метод `Promise.race()`, инициирующий выполнение целого ряда длительных операций?

6.1. Достижение изящности асинхронного кода с помощью генераторов и обещаний

Представьте себя в роли разработчика, работающего на сайте `freelanceninja.com`, где посетители могут набирать вольнонаемных нинздя

[1] Здесь было бы уместно использовать термин *обязательство*, как наиболее логичный и понятный в данной ситуации, т.е. обязательство что-то выполнить в коде в будущем. Собственно обязательство – это и есть одно из значений английского термина *promises*. Однако те, кто переводил русскую документацию для MDN, решили использовать термин обещания, как первое значение слова promises в английском словаре ☺. Поэтому, чтобы не вносить путаницу, в книге мы будем пользоваться *обещаниями*. – *Примеч. ред.*

для выполнения тайных заданий. Ваша задача – реализовать функциональные средства, дающие пользователям возможность получать подробные сведения о наиболее трудных заданиях, выполненных самыми известными нинзя. Данные, представляющие нинзя, сводки их заданий, а также подробности их выполнения хранятся на удаленном сервере в формате JSON. Для решения стоящей перед вами задачи вы можете написать код, подобный следующему:

```
try {
  var ninjas = syncGetJSON("ninjas.json");
  var missions = syncGetJSON(ninjas[0].missionsUrl);
  var missionDetails = syncGetJSON(missions[0].detailsUrl);
  // изучить описание задания
}
catch(e){
  // Какая досада! Получить подробности задания не удалось!
}
```

Понять приведенный выше фрагмент кода нетрудно, и если возникнет ошибка на любой стадии выполнения, ее нетрудно перехватить и обработать в блоке оператора `catch`. Но, к сожалению, этот код страдает серьезным недостатком. Получение данных от сервера – длительная операция, а поскольку в JavaScript используется однопоточная модель выполнения кода, то пользовательский интерфейс окажется заблокированным до момента завершения длительной операции. В итоге приложение не будет оперативно реагировать на действия пользователя, вызывая у него разочарование. Чтобы устранить подобный недостаток, можно переписать рассматриваемый здесь код, делая в нем обратные вызовы по окончании операции без блокировки пользовательского интерфейса:

```
getJSON("ninjas.json", function(err, ninjas){
  if(err) {
    console.log("Error fetching list of ninjas", err);
    return;
  }
  getJSON(ninjas[0].missionsUrl, function(err, missions) {
    if(err) {
      console.log("Error locating ninja missions", err);
      return;
    }
    getJSON(missions[0].detailsUrl, function(err, missionDetails){
      if(err) {
        console.log("Error locating mission details", err);
        return;
      }
      // изучить данные разведки
    });
  });
});
```

И хотя действие приведенного выше кода будет намного лучше восприниматься пользователями, согласитесь, что он содержит немало шаблонного кода для обработки ошибок и выглядит неопрятно и неприглядно. И здесь на помощь приходят генераторы и обещания. Сочетая их вместе, можно превратить не блокирующийся, но неуклюжий код обратных вызовов в следующий намного более изящный код:

Функция-генератор обозначается знаком звездочки, указываемым после ключевого слова `function`.
В функциях-генераторах можно применять новое ключевое слово `yield`

Обещания скрываются в методе `getJSON()`

```
async(function*(){
  try {
    const ninjas = yield getJSON("ninjas.json");
    const missions = yield getJSON(ninjas[0].missionsUrl);
    const missionDescription = yield getJSON(missions[0].detailsUrl);
    // изучить подробности задания
  }
  catch(e) {
    // получить подробности задания не удалось
  }
});
```

Если приведенный выше фрагмент кода не совсем понятен, а употребляемый в нем синтаксис (например, `function*` или `yield`) вам незнаком, не отчаивайтесь — к концу этой главы вы сможете восполнить пробел в своих знаниях. А до тех пор просто сравните изящество этого не блокирующегося кода, где применяются генераторы и обещания, с неуклюжестью предыдущего кода обратных вызовов.

Итак, приступим потихоньку к исследованию функций-генераторов — первой вехи на пути к изящному асинхронному коду.

6.2. Использование функций-генераторов

Генераторы являются совершенно новым типом функций, существенно отличаясь от стандартных, обыкновенных функций. *Генератор* — это функция, генерирующая последовательность значений, но не сразу, как обычная функция, а по запросу. У генератора приходится запрашивать новое значение, и в ответ генератор выдает это значение или уведомляет, что у него нет больше значений для генерирования. Но еще любопытнее, что после того, как значение будет сгенерировано, функция-генератор не завершает работу, как обычно, а лишь приостанавливает свое выполнение. Когда же поступает запрос очередного значения, функция-генератор возобновляет свое выполнение там, где оно было приостановлено.

В листинге 6.1 демонстрируется простой пример применения функции-генератора для формирования последовательного ряда вооружений.

Сначала определяется генератор, формирующий последовательный ряд вооружений. Чтобы создать функцию-генератор, достаточно указать знак звездочки (`*`) после ключевого слова `function`. Это дает возможность воспользоваться новым ключевым словом `yield` в теле функции-генератора для получе-

ния отдельных значений. Синтаксис функции-генератора наглядно демонстрируется на рис. 6.1.

Листинг 6.1. Формирование последовательности значений с помощью функции-генератора

```
function* WeaponGenerator(){
yield "Katana";
yield "Wakizashi";
yield "Kusarigama";
}

for(let weapon of WeaponGenerator()) {
  assert(weapon !== undefined, weapon);
}
```

Определить функцию-генератор, указав знак * после ключевого слова function

Сгенерировать отдельные значения, используя ключевое слово yield

Употребить сгенерированные значения в новом цикле for-of

В данном примере кода создается функция-генератор `WeaponGenerator()`, формирующая последовательный ряд значений, обозначающих разные виды вооружений: `Katana`, `Wakizashi` и `Kusarigama`. Употребить этот последовательный ряд вооружений можно, например, в новом цикле `for-of`, как показано ниже.

```
for(let weapon of WeaponGenerator()) {
  assert(weapon, weapon);
}
```

Результат вызова функции-генератора в этом цикле `for-of` приведен на рис. 6.2. (Более подробно цикл `for-of` рассматривается далее в этой главе.)

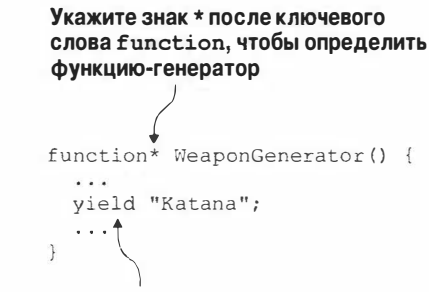


Рис. 6.1. Чтобы определить функцию-генератор, достаточно указать знак звездочки (*) после ключевого слова `function`

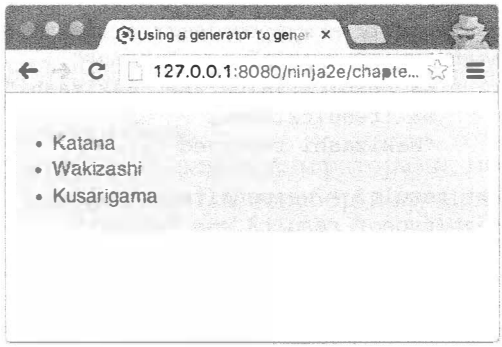


Рис. 6.2. Результат циклического обращения к функции-генератору `WeaponGenerator()`

В правой части цикла `for-of` указывается результат вызова функции-генератора. Но если внимательно присмотреться к телу функции-генератора `WeaponGenerator()`, то можно заметить, что в нем отсутствует оператор

return. Не означает ли это, что в правой части цикла for-of должно быть вычислено неопределенное значение undefined, как это обычно происходит со стандартной функцией?

Дело в том, что генераторы совершенно не похожи на стандартные функции. В частности, вызов функции-генератора приводит не к ее выполнению, а к созданию объекта, называемого *итератором*. Исследуем далее этот объект.

6.2.1. Управление генератором с помощью итератора

Вызов генератора совсем не означает, что тело функции-генератора будет выполняться. Вместо этого создается итератор — объект, через который можно взаимодействовать с генератором. С помощью итератора можно, например, запрашивать дополнительные значения. Рассмотрим принцип действия итератора на примере кода из листинга 6.2, который является переделанным с данной целью вариантом предыдущего примера кода.

Листинг 6.2. Управление генератором с помощью итератора

```
function* WeaponGenerator(){
  yield "Katana";
  yield "Wakizashi";
}

const weaponsIterator = WeaponGenerator();

const result1 = weaponsIterator.next();
assert(typeof result1 === "object"
       && result1.value === "Katana"
       && !result1.done,
       "Katana received!");

const result2 = weaponsIterator.next();
assert(typeof result2 === "object"
       && result2.value === "Wakizashi"
       && !result2.done,
       "Wakizashi received!");

const result3 = weaponsIterator.next();
assert(typeof result3 === "object"
       && result3.value === undefined
       && result3.done,
       "There are no more results!");
```

Определить генератор, создающий последовательный ряд вооружений

В результате вызова генератора создается итератор, с помощью которого можно управлять выполнением генератора

При вызове метода next() итератора запрашивается новое значение из генератора

В итоге получается объект с возвращаемым значением, а также признак того, что у генератора еще имеются значения

В результате следующего вызова получается еще одно значение из генератора

Когда не остается больше кода для выполнения, генератор возвращает значение undefined, указывая на свое завершение

При вызове генератора создается новый *итератор*, как показано ниже.

```
const weaponsIterator = WeaponGenerator();
```

Итератор служит для управления выполнением генератора. Одна из главных особенностей объекта-итератора состоит в том, что у него есть метод next(), с помощью которого можно управлять генератором, запрашивая у него значение следующим образом:

```
const result1 = weaponsIterator.next();
```

В ответ на этот вызов генератор выполняет свой код до тех пор, пока не будет достигнуто ключевое слово `yield`, где получается промежуточный результат (т.е. один элемент из генерируемой последовательности элементов) и возвращается *новый* объект, инкапсулирующий этот результат и сообщающий, завершена ли работа генератора или еще нет.

Как только будет получено текущее значение, генератор приостановит свое выполнение, перейдя без блокировки в режим ожидания запроса другого значения. Это довольно эффективное языковое средство, которым не обладают стандартные функции, и мы еще не раз выгодно воспользуемся им в дальнейшем.

В данном случае при первом вызове метода `next()` итератора код генератора выполняется вплоть до первого выражения `yield "Katana"`. При этом возвращается объект с установленным значением `"Katana"` свойства `value` и логическим значением `false` свойства `done`, указывающим на то, что еще остались значения для генерирования.

Далее запрашивается еще одно значение из генератора. С этой целью делается еще один вызов метода `next()` объекта `weaponIterator`, как показано ниже.

```
const result2 = weaponsIterator.next();
```

В итоге генератор возобновляет выполнение там, где оно было приостановлено в прошлый раз, и выполняет свой код до тех пор, пока не будет получено очередное промежуточное значение в выражении `yield "Wakizashi"`. После этого генератор приостанавливает свое работу и возвращает объект, содержащий значение `"Wakizashi"`.

И, наконец, когда метод `next()` вызывается в третий раз, генератор также возобновляет свое выполнение. Но на этот раз у него отсутствует код для выполнения, поэтому генератор возвращает объект с установленным значением `undefined` свойства `value` и логическим значением `true` свойства `done`, указывающим на то, что генератор завершил свою работу.

Итак, выяснив, каким образом итераторы управляют генераторами, можно исследовать далее, каким образом осуществляется перебор получаемых значений.

Обход итератора

У итератора, создаваемого при вызове генератора, существует метод `next()`, с помощью которого можно запрашивать очередное значение у генератора. Метод `next()` возвращает объект, содержащий значение, вычисленное генератором, а также информацию, хранящуюся в свойстве `done` и указывающую на возможность получения дополнительных значений.

Мы воспользуемся возможностями итератора, чтобы перебрать значения, производимые генератором, в старом добром цикле `while`, как демонстрируется в примере кода из листинга 6.3.

Листинг 6.3. Перебор результатов, возвращаемых генератором, в цикле `while`

```
function* WeaponGenerator(){
  yield "Katana";
  yield "Wakizashi";
}

const weaponsIterator = WeaponGenerator();
let item;
while(!(item = weaponsIterator.next()).done) {
  assert(item !== null, item.value);
}
```

Создать итератор

Создать переменную для хранения элементов генерируемой последовательности

На каждом шаге цикла из генератора извлекается и выводится одно значение. Перебор значений прекращается, когда в генераторе исчерпаны значения для генерирования

И в данном примере создается объект-итератор, для чего функция-генератор вызывается следующим образом:

```
const weaponsIterator = WeaponGenerator();
```

Кроме того, создается переменная `item`, в которой предполагается хранить отдельные значения, производимые генераторами. А далее организуется цикл `while` с несколько усложненным условием выполнения, которое требует дополнительного пояснения.

```
while(!(item = weaponsIterator.next()).done) {
  assert(item !== null, item.value)
}
```

На каждом шаге данного цикла вызывается метод `next()` для объекта `weaponsIterator`, чтобы извлечь из генератора очередное значение, которое сохраняется в переменной `item`. Как и у всех подобных объектов, у объекта, на который ссылается переменная `item`, имеется свойство `done`, указывающее, окончил ли генератор производить значения. Если генератор еще не закончил свою работу, то выполняется еще один шаг цикла, а иначе цикл завершается.

Именно таким образом и действует цикл `for-of` из представленного ранее первого примера генератора. Цикл `for-of` предоставляет синтаксическое удобство для обхода итераторов, как показано ниже.

```
for(var item of WeaponGenerator ()){
  assert(item !== null, item);
}
```

Вместо того чтобы вызывать вручную метод `next()` соответствующего итератора и постоянно проверять, завершилось ли генерирование значений, можно воспользоваться циклом `for-of`, который делает то же самое, но только автоматически.

Поручение выполнения другому генератору

Из одной стандартной функции нередко вызывается другая стандартная функция. Аналогичным образом иногда требуется поручить выполнение одно-

го генератора другому. Рассмотрим в качестве примера код из листинга 6.4, где генерируются имена воинов и ниндзя.

Листинг 6.4. Поручение выполнения другому генератору с помощью операции `yield*`

```
function* WarriorGenerator(){
  yield "Sun Tzu";
  yield* NinjaGenerator();
  yield "Genghis Khan";
}

function* NinjaGenerator(){
  yield "Hattori";
  yield "Yoshi";
}

for(let warrior of WarriorGenerator()){
  assert(warrior !== null, warrior);
}
```

В операции `yield*` выполнение поручается другому генератору

Если выполнить код из листинга 6.4, то в конечном итоге будет выведен такой результат: `Sun Tzu`, `Hattori`, `Yoshi`, `Genghis Khan`. Формирование имени `Sun Tzu`, вероятно, не должно удивлять. Ведь это первое значение, произведенное генератором `WarriorGenerator()`. Но особого внимания заслуживает второе выводимое имя `Hattori`.

В операции `yield*` выполнение поручается другому генератору. В данном примере выполнение поручается текущим генератором `WarriorGenerator()` новому генератору `NinjaGenerator()`, а все вызовы метода `next()` итератора текущего генератора `WarriorGenerator()` перенаправляются новому генератору `NinjaGenerator()` до тех пор, пока он не завершит свою работу. Поэтому в данном примере после имени `Sun Tzu` генерируются имена `Hattori` и `Yoshi`. И лишь тогда, когда генератор `NinjaGenerator()` завершит свою работу, продолжит свое выполнение исходный генератор, выводя имя `Genghis Khan`. Следует заметить, что все это происходит прозрачно для кода, где вызывается исходный генератор. Для цикла `for-of` совершенно не важно, что исходный генератор `WarriorGenerator()` поручает выполнение другому генератору. Вызов метода `next()` продолжается в нем до тех пор, пока выполнение не завершится.

Итак, рассмотрев принцип действия генераторов в целом и порядок поручения работы одного генератора другому, перейдем к некоторым примерам применения генераторов.

6.2.2. Применение генераторов

Возможность генерировать последовательности элементов, конечно, замечательна и прекрасна, но обратимся к более практическим примерам, начиная с простого случая генерирования уникальных идентификаторов.

Генерирование уникальных идентификаторов с помощью генераторов

При создании некоторых объектов нередко возникает потребность присвоить уникальный идентификатор каждому объекту. Для этого проще всего воспользоваться глобальной переменной-счетчиком, несмотря на то, что это довольно ненадежное решение, поскольку значение такой переменной может быть случайно запорчено где-нибудь в прикладном коде. Другая возможность состоит в применении генератора, как демонстрируется в примере кода из листинга 6.5.

Листинг 6.5. Генерирование уникальных идентификаторов с помощью генераторов

```
function *IdGenerator(){
  let id = 0;
  while(true){
    yield ++id;
  }
}

const idIterator = IdGenerator();

const ninja1 = { id: idIterator.next().value };
const ninja2 = { id: idIterator.next().value };
const ninja3 = { id: idIterator.next().value };

assert(ninja1.id === 1, "First ninja has id 1");
assert(ninja2.id === 2, "Second ninja has id 2");
assert(ninja3.id === 3, "Third ninja has id 3");
```

Определить функцию-генератор IdGenerator()

Переменная, в которой отслеживаются идентификаторы. Эту переменную нельзя видоизменить за пределами генератора

Цикл, в котором формируется бесконечная последовательность идентификаторов

Итератор, через который у генератора запрашиваются новые идентификаторы

Запросить три новых идентификатора

Проверить, все ли прошло нормально

Сначала в данном примере кода определяется функция-генератор с одной локальной переменной `id`, представляющей счетчик идентификаторов. А поскольку переменная `id` является локальной для генератора, то можно не опасаться, что кто-нибудь случайно видоизменит ее где-нибудь еще в коде. Далее организуется цикл `while`, на каждом шаге которого получается новое значение идентификатора `id` и выполнение генератора приостанавливается до тех пор, пока не поступит запрос очередного идентификатора:

```
function *IdGenerator(){
  let id = 0;
  while(true){
    yield ++id;
  }
}
```

Примечание

Как правило, бесконечные циклы в стандартных функциях не организуются, но в функциях-генераторах они вполне уместны! Всякий раз, когда в генераторе встречается оператор **`yield`**, выполнение генератора приостанавливается до

тех пор, пока не будет вызван метод **`next()`**. Таким образом, при каждом вызове метода **`next()`** выполняется лишь один шаг бесконечного цикла, на котором отсылается обратно следующее значение идентификатора.

После определения генератора создается объект-итератор, как показано ниже.

```
const idIterator = IdGenerator();
```

Это дает возможность управлять генератором, вызывая метод `idIterator.next()`. Выполнение генератора продолжается до тех пор, пока не встретится оператор `yield`, возвращающий новое значение идентификатора, которым можно воспользоваться во вновь создаваемых объектах следующим образом:

```
const ninja1 = { id: idIterator.next().value };
```

Как видите, все довольно просто. Вместо ненадежных глобальных переменных, значения которых могут быть случайно изменены, в данном случае применяется итератор для запрашивания значений у генератора. А если в дальнейшем для отслеживания идентификаторов (например, `samurai`) потребуется еще один итератор, то для этой цели можно инициализировать новый генератор.

Обход модели DOM с помощью генераторов

Как пояснялось в главе 2, компоновка веб-страницы организуется на основе модели DOM — древовидной структуры HTML-узлов, где у каждого узла, кроме корневого, имеется только один родительский узел и может быть от нуля и больше дочерних узлов. В связи с тем что модели DOM принадлежит столь важная роль основополагающей структуры в веб-разработке, немалая часть прикладного кода отводится для обхода узлов этой модели. С этой целью можно, например, относительно просто реализовать рекурсивную функцию, которая будет выполняться при обходе каждого узла, как демонстрируется в примере кода из листинга 6.6.

Листинг 6.6. Рекурсивный обход модели DOM

```
<div id="subTree">
  <form>
    <input type="text"/>
  </form>
  <p>Paragraph</p>
  <span>Span</span>
</div>
<script>
  function traverseDOM(element, callback) {
    callback(element);
    element = element.firstElementChild;
    while (element) {
      traverseDOM(element, callback);
      element = element.nextElementSibling;
    }
  }
```

Обработать текущий узел посредством обратного вызова

Обойти каждый порожденный элемент модели DOM

```
  const subTree = document.getElementById("subTree");
  traverseDOM(subTree, function(element) {
    assert(element !== null, element.nodeName);
  });
</script>
```

Начать весь процесс, вызвав функцию `traverseDOM()` для корневого элемента

В данном примере кода для обхода всех элементов, порожденных элементом с идентификатором `subtree`, служит рекурсивная функция, вызываемая в процессе регистрации каждого типа узла, посещаемого при обходе. В коде из данного примера выводятся узлы `DIV`, `FORM`, `INPUT`, `P` и `SPAN`.

Нам не раз приходилось писать такой код для обхода узлов модели DOM, и он вполне нас устраивал. Но теперь, когда в нашем распоряжении имеются генераторы, мы можем поступить иначе, как показано в примере кода из листинга 6.7.

Листинг 6.7. Обход модели DOM с помощью генераторов

```
function* DomTraversal(element){
  yield element;
  element = element.firstElementChild;
  while (element) {
    yield* DomTraversal(element);
    element = element.nextElementSibling;
  }
}

const subTree = document.getElementById("subTree");
for(let element of DomTraversal(subTree)) {
  assert(element !== null, element.nodeName);
}
```

Применить операцию `yield*`, чтобы передать управление циклическим обходом экземпляру генератора `DomTraversal()`

Обойти узлы в цикле `for-of`

В примере кода из листинга 6.7 наглядно показано, что обход узлов модели DOM с помощью генераторов организуется так же просто, как и посредством стандартной рекурсии, но у такого способа имеется дополнительное преимущество, исключающее применение не очень изящного синтаксиса обратных вызовов. Вместо обработки поддерева каждого посещаемого узла путем рекурсивного перехода на другой уровень достаточно создать одну функцию-генератор для каждого посещаемого узла и поручить ей выполнение. Это дает возможность написать принципиально рекурсивный код итерационным способом. Преимущество такого способа заключается в том, что мы можем получить генерируемую последовательность узлов в простом цикле `for-of`, не прибегая к неудобным обратным вызовам.

Рассматриваемый здесь пример особенно примечателен тем, что он наглядно показывает также, как пользоваться генераторами, чтобы отделить код, производящий значения (в данном случае HTML-узлы), от кода, использующего последовательность генерируемых значений (в данном случае цикл `for-of`, где регистрируются посещаемые узлы), не прибегая к обратным вызовам. Кроме

того, применение циклов в некоторых случаях оказывается намного более естественным, чем рекурсии, и поэтому неплохо всегда иметь разные варианты выполнения одной и той же задачи.

После рассмотрения некоторых практических примеров применения генераторов вернемся к несколько более теоретическому вопросу, чтобы выяснить, каким образом происходит обмен данными с выполняющимся генератором.

6.2.3. Обмен данными с генератором

В представленных до сих пор примерах кода было показано, как возвратить несколько значений из генератора с помощью выражения `yield`. Но генераторы можно использовать намного более эффективно! В частности, генераторам можно посылать данные, устанавливая двухстороннюю связь с ними, получать с их помощью промежуточные результаты, используя последние в расчетах за пределами генератора, а затем посылать обратно генератору совершенно новые данные и возобновлять его выполнение. Мы воспользуемся такой возможностью в конце главы для создания эффективного асинхронного кода, а пока рассмотрим более простые примеры обмена данными с генератором.

Отправка значений в виде аргументов функции-генератора

Для отправки данных генератору его проще всего рассматривать как любую другую функцию, которой при вызове можно передать аргументы. Рассмотрим в качестве примера код из листинга 6.8.

Листинг 6.8. Отправка и получение данных из генератора

```
function* NinjaGenerator(action) {
  const imposter = yield ("Hattori " + action);

  assert(imposter === "Hanzo",
         "The generator has been infiltrated");
  yield ("Yoshi (" + imposter + ") " + action);
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
assert(result1.value === "Hattori skulk", "Hattori is skulking");

const result2 = ninjaIterator.next("Hanzo");
assert(result2.value === "Yoshi (Hanzo) skulk",
       "We have an imposter!");
```

Значение, посылаемое через метод `next()`, становится значением выражения `yield`, и поэтому самозванца зовут Ханзо

Генератору можно передавать обычные аргументы, как и любой другой функции

Произведя значение, генератор, как по волшебству, возвращает промежуточный результат вычислений. Вызвав метод `next()` итератора с аргументом, можно отправить данные обратно генератору

Обычная передача аргумента

Начать выполнение генератора и проверить, получено ли правильное значение

Отправить данные генератору в качестве аргумента метода `next()` и проверить, правильно ли передано значение

В функции, получающей данные, нет ничего особенного. Ведь то же самое делают и все стандартные функции. Но не следует забывать, что, обладая столь примечательной особенностью, генераторы могут приостанавливать и возобновлять свою работу. И в отличие от стандартных функций, генераторы могут получать данные *даже* после того, как было начато их выполнение — в момент возобновления работы при поступлении запроса на следующее значение.

Отправка значений генератору через метод `next()`

Помимо предоставления данных при первом вызове генератора, имеется также возможность посылать данные генератору, передав их в качестве аргументов методу `next()`. В ходе этого процесса возобновляется выполнение генератора. Переданное значение используется генератором как значение того выражения `yield`, на котором генератор приостановил свое выполнение в прошлый раз (рис. 6.3).

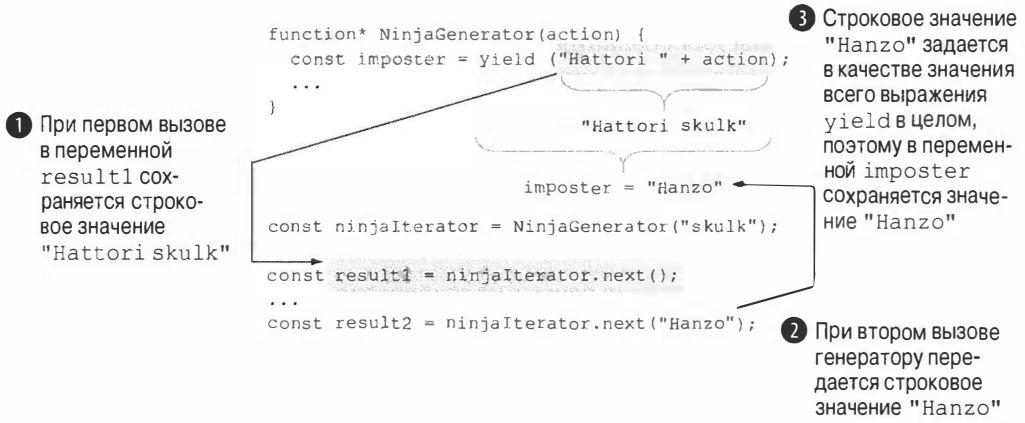


Рис. 6.3. При первом вызове метода **`ninjaIterator.next()`** запрашивается новое значение у генератора, который возвращает строковое значение **`"Hattori skulk"`** и приостанавливает свое выполнение на выражении **`yield`**. Новое значение запрашивается и при втором вызове **`ninjaIterator.next("Hanzo")`**, но на этот раз генератору передается также аргумент со значением **`"Hanzo"`**. Это значение будет использовано в качестве значения всего выражения **`yield`**, а в переменной **`imposter`** сохранится значение **`"Hanzo"`**

В данном примере делаются два вызова метода `next()` объекта `ninjaIterator`. При первом вызове метода `ninjaIterator.next()` у генератора запрашивается первое значение. А поскольку выполнение генератора еще не началось, то данный вызов приводит к запуску генератора, который сначала вычисляет выражение `"Hattori " + action`, создавая строковое значение `"Hattori skulk"`, а затем приостанавливает свое выполнение. Но в этом нет ничего особенного. Ведь в предыдущих примерах кода это делалось уже не раз в данной главе.

Нечто более интересное происходит при втором вызове метода `ninjaIterator.next("Hanzo")`. На этот раз метод `next()` используется для пере-

дачи данных обратно генератору. Функция-генератор терпеливо ожидает, приостановив свое выполнение на выражении `yield ("Hattori " + action)`, а строковое значение `"Hanzo"`, переданное в качестве аргумента методу `next()`, служит в качестве значения всего выражения `yield`. В данном случае это означает, что в выражении `imposter = yield ("Hattori " + action)` переменной `imposter` будет присвоено значение `"Hanzo"`.

Именно таким образом достигается двухсторонний обмен данными с генератором. В частности, для возврата данных из генератора служит выражение `yield`, а для передачи данных обратно генератору — метод `next()` объекта-итератора.

Примечание

Метод **`next()`** передает значение ожидающему своей очереди выражению **`yield`**, и если такие выражения отсутствуют, то ничего и не передается. Именно поэтому значения нельзя передать при первом вызове метода **`next()`**. Но не следует забывать, что если генератору требуется предоставить первоначальное значение, то это можно сделать при вызове самого генератора, как, например, **`NinjaGenerator("skulk")`**.

Генерирование исключений

Имеется еще один, немного нетрадиционный способ передачи значения генератору, который состоит в генерировании исключения. Помимо метода `next()`, у каждого итератора имеется также метод `throw()`, которым можно воспользоваться, чтобы сгенерировать исключение в генераторе. Обратимся снова к примеру кода, приведенному в листинге 6.9.

Листинг 6.9. Генерирование исключений в генераторе

```
function* NinjaGenerator() {
  try{
    yield "Hattori";
    fail("The expected exception didn't occur");
  }
  catch(e){
    assert(e === "Catch this!", "Aha! We caught an exception");
  }
}

const ninjaIterator = NinjaGenerator();

const result1 = ninjaIterator.next();
assert(result1.value === "Hattori", "We got Hattori");

ninjaIterator.throw("Catch this!");
```

Этот метод **`fail()`** не должен быть достигнут

Перехватить исключения и проверить, получено ли предполагаемое исключение

Извлечь одно значение из генератора

Сгенерировать исключение в генераторе

Исходный код из листинга 6.9 начинается подобно коду из листинга 6.8 с определения функции-генератора `NinjaGenerator()`. Но на этот раз тело

функции-генератора несколько иное. В частности, оно полностью заключено в блок операторов try/catch, как показано ниже.

```
function* NinjaGenerator() {
  try{
    yield "Hattori";
    fail("The expected exception didn't occur");
  }
  catch(e){
    assert(e === "Catch this!", "Aha! We caught an exception");
  }
}
```

Далее создается итератор и получается одно значение из генератора:

```
const ninjaIterator = NinjaGenerator();
const result1 = ninjaIterator.next();
```

И, наконец, метод throw(), имеющийся у всех итераторов, вызывается с целью сгенерировать исключение в генераторе:

```
ninjaIterator.throw("Catch this!");
```

Если выполнить код из листинга 6.9, то можно обнаружить, что исключение генерируется именно так, как и предполагалось (рис. 6.4).

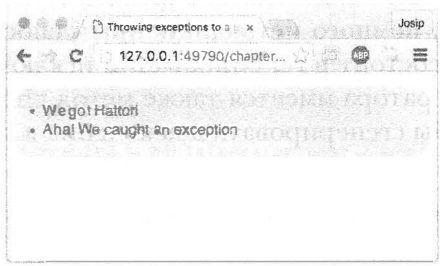


Рис. 6.4. Исключения в генераторе можно инициировать из вне

На первый взгляд, возможность генерировать исключения в генераторе может показаться немного странной. Зачем вообще это делать? Не беспокойтесь, мы не собираемся держать вас в полном неведении. В конце этой главы мы воспользуемся такой возможностью, чтобы усовершенствовать асинхронный обмен данными с сервером. Так что потерпите немного.

А теперь, рассмотрев ряд особенностей генераторов, перейдем к исследованию внутреннего механизма их действия.

6.2.4. Исследование внутреннего механизма действия генераторов

Мы выяснили, что вызов генератора приводит не к его выполнению, а к созданию нового итератора, с помощью которого можно запрашивать значения

у генератора. И как только генератор произведет (или сформирует) значение, он приостановит свое выполнение в ожидании следующего запроса. Таким образом, генератор действует в какой-то степени подобно небольшой программе, а точнее — конечному автомату, переходящему из одного состояния в другое, как поясняется ниже.

- **Запуск с приостановкой.** Это начальное состояние генератора, в котором он оказывается при своем создании. Код самого генератора не выполняется.
- **Выполнение.** В этом состоянии выполняется код генератора. Выполнение начинается с самого начала или продолжается с того места, где генератор был приостановлен в прошлый раз. Генератор переходит в это состояние, когда вызывается метод `next()` соответствующего итератора и еще имеется код для выполнения.
- **Выдача с приостановкой.** Когда генератор достигает выражения `yield` в ходе своего выполнения, он создает новый объект, содержащий возвращаемое значение, выдает его и приостанавливает свое выполнение. В этом состоянии генератор находится в режиме ожидания поступления следующего запроса на выполнение.
- **Завершение.** Генератор перейдет в это состояние, если в ходе своей работы он встретит оператор `return` или исчерпает весь код для выполнения.

Перечисленные выше состояния генератора наглядно показаны на рис. 6.5.

А теперь перейдем к еще более углубленному исследованию, чтобы выяснить, каким образом выполнение генераторов отслеживается с помощью контекстов выполнения.

Слежение за генераторами с помощью контекстов выполнения

В предыдущей главе был представлен контекст выполнения — внутренний интерпретатор JavaScript, применяемый для слежения за выполнением функций. И несмотря на все особенности генераторов, они по-прежнему остаются функциями. Поэтому исследуем более подробно их взаимосвязь с контекстами выполнения. И начнем мы с приведенного ниже простого фрагмента кода. В этом фрагменте кода повторно используется генератор, который создает следующие строковые значения: `"Hattori skulk"` и `"Yoshi skulk"`.

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");
const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

```
function* NinjaGenerator(){
    yield "Hattori";
    yield "Yoshi";
}
```

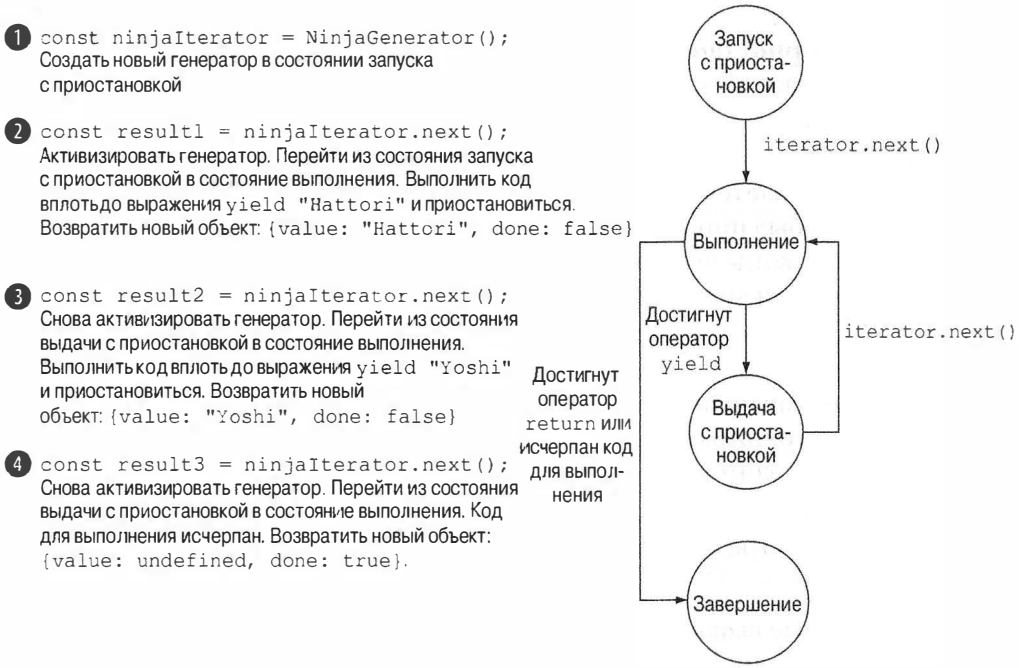


Рис. 6.5. В ходе выполнения генератор меняет состояния под воздействием вызовов метода **`next()`** соответствующего итератора

Исследуем состояние этого фрагмента прикладного кода, а также стек контекстов выполнения в различных точках его выполнения. На рис. 6.6 приведены моментальные снимки, сделанные в двух местах выполнения прикладного кода. Первый моментальный снимок демонстрирует состояние выполнения прикладного кода *до* вызова функции `NinjaGenerator()` ❶. А поскольку выполняется глобальный код, то стек контекстов выполнения содержит только глобальный контекст выполнения со ссылкой на глобальную среду, в которой хранятся идентификаторы. Функцию обозначает только идентификатор `NinjaGenerator`, тогда как значения всех остальных идентификаторов не определены (`undefined`).

При следующем вызове функции `NinjaGenerator()` ❷:

```
const ninjaIterator = NinjaGenerator("skulk");
```

поток управления программой переходит к выполнению генератора и, как это обычно происходит с любой другой функцией, при этом создается и размещается в стеке новый элемент контекста выполнения функции `NinjaGenerator()` (вместе с соответствующей лексической средой). Но поскольку генераторы

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

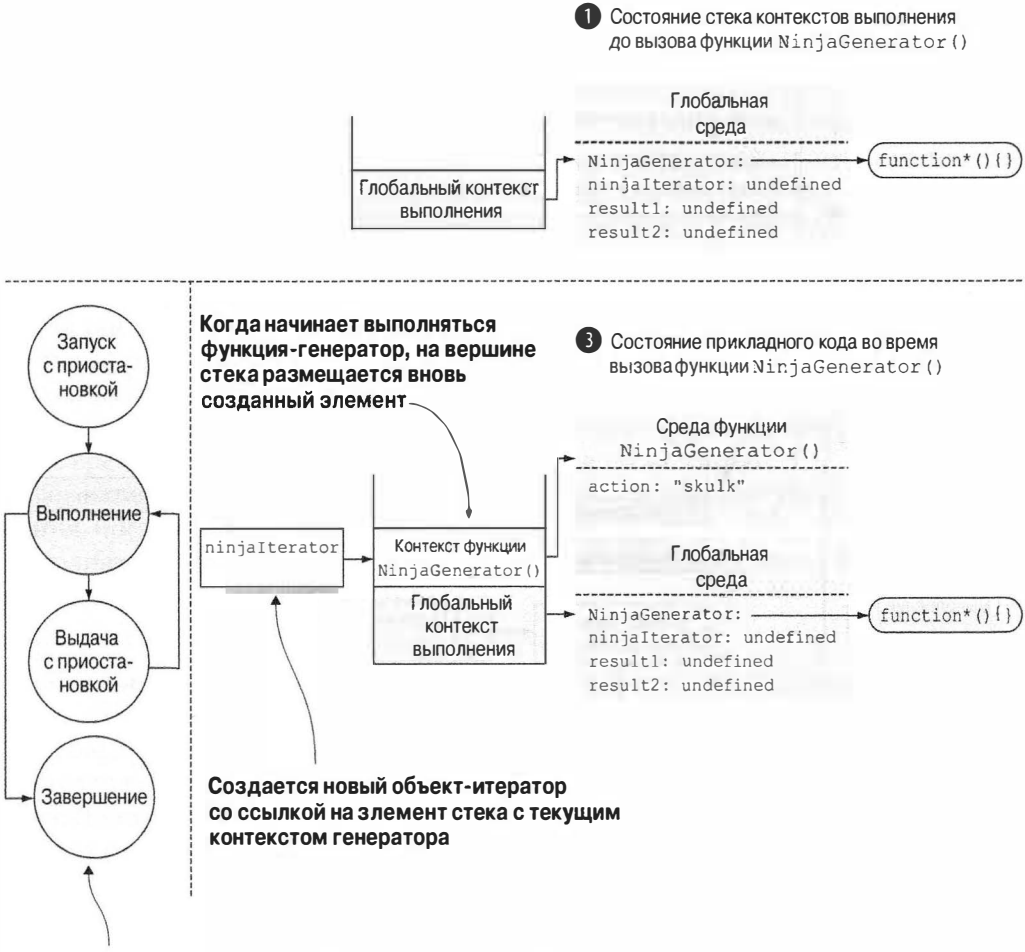


Рис. 6.6. Состояние стека контекстов выполнения *до* вызова функции **`NinjaGenerator()`** ❶ и во время вызова функции **`NinjaGenerator()`** ❷

относятся к категории специальных функций, то код функции вообще *не* выполняется. Вместо этого создается и возвращается новый итератор, к которому предстоит далее обращаться в прикладном коде по ссылке `ninjaIterator`.

Итератор служит для управления выполнением генератора, и поэтому он получает ссылку на тот контекст выполнения, в котором был создан.

Но самое интересное происходит именно тогда, когда поток управления программой покидает генератор (рис. 6.7). Как правило, когда выполнение программы возвращается из обычной функции, соответствующий контекст выполнения удаляется из стека и полностью теряется. Но совсем иначе дело обстоит с генераторами.

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

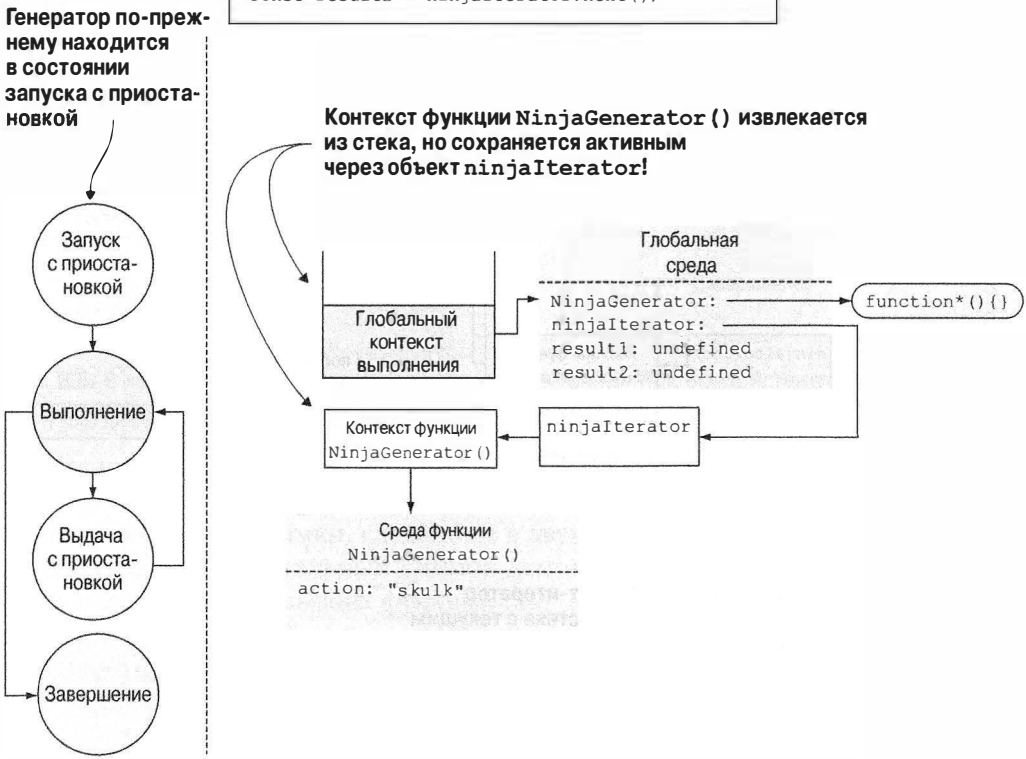


Рис. 6.7. Состояние прикладного кода при возврате из вызываемой функции `NinjaGenerator()`

Из стека извлекается соответствующий элемент `NinjaGenerator`, но он *не* теряется, поскольку в объекте `ninjaIterator` сохраняется ссылка на него. Как видите, это похоже на замыкания. Ведь в замыканиях необходимо поддерживать в активном состоянии переменные, существующие на момент определения функции, и поэтому в функциях сохраняется ссылка на ту среду, где они

были созданы. Благодаря этому можно убедиться, что среда и ее переменные активны до тех пор, пока действует сама функция. С другой стороны, генераторы должны уметь возобновлять свое выполнение. А поскольку выполнение всех функций отслеживается в контекстах выполнения, то в итераторе хранится ссылка на его контекст выполнения, а следовательно, он остается активным до тех пор, пока он нужен итератору.

Кое-что интересное происходит и в тот момент, когда для итератора вызывается метод `next()`:

```
const result1 = ninjaIterator.next();
```

Если бы это был вызов стандартной, обыкновенной функции, он привел бы к созданию нового элемента контекста выполнения метода `next()` и последующему его размещению в стеке. Но генераторы, как вы, должно быть, уже заметили, существенно отличаются от обыкновенных функций, и поэтому при вызове метода `next()` для итератора дело обстоит совершенно иначе: снова активизируется соответствующий контекст выполнения (в данном случае контекст функции `NinjaGenerator()`), который размещается на вершине стека, а выполнение продолжается с того места, где оно было приостановлено (рис. 6.8).

На рис. 6.8 наглядно показано коренное отличие стандартных функций от функций-генераторов. С одной стороны, стандартные функции могут вызываться только заново, и при каждом их вызове создается *новый* контекст выполнения. А с другой стороны, контекст выполнения функции-генератора может быть временно приостановлен и возобновлен по желанию.

В рассматриваемом здесь примере генератор переходит в состояние выполнения, поскольку это первый вызов метода `next()` и генератор еще не начинал выполняться. Следующий интересный момент наступает, когда функция-генератор достигает выражения:

```
yield "Hattori " + action
```

Когда интерпретатор достигает ключевого слова `yield` вычисляется указанное в его правой части строковое выражение, которое приобретает значение `"Hattori skulk"`. Это означает, что строковое значение `"Hattori skulk"` является первым промежуточным результатом выполнения генератора и что его выполнение требуется приостановить, возвратив данное значение. С точки зрения состояния прикладного кода происходит то же самое, что и прежде: контекст функции `NinjaGenerator()` извлекается из стека, но он не теряется, так как ссылка на него сохраняется в объекте `ninjaIterator`. В настоящий момент генератор приостанавливается и переходит в состояние выдачи с приостановкой без блокировки. А выполнение программы возобновляется в глобальном коде, где произведенное и выданное генератором значение сохраняется в переменной `result1`. Текущее состояние прикладного кода наглядно показано на рис. 6.9.

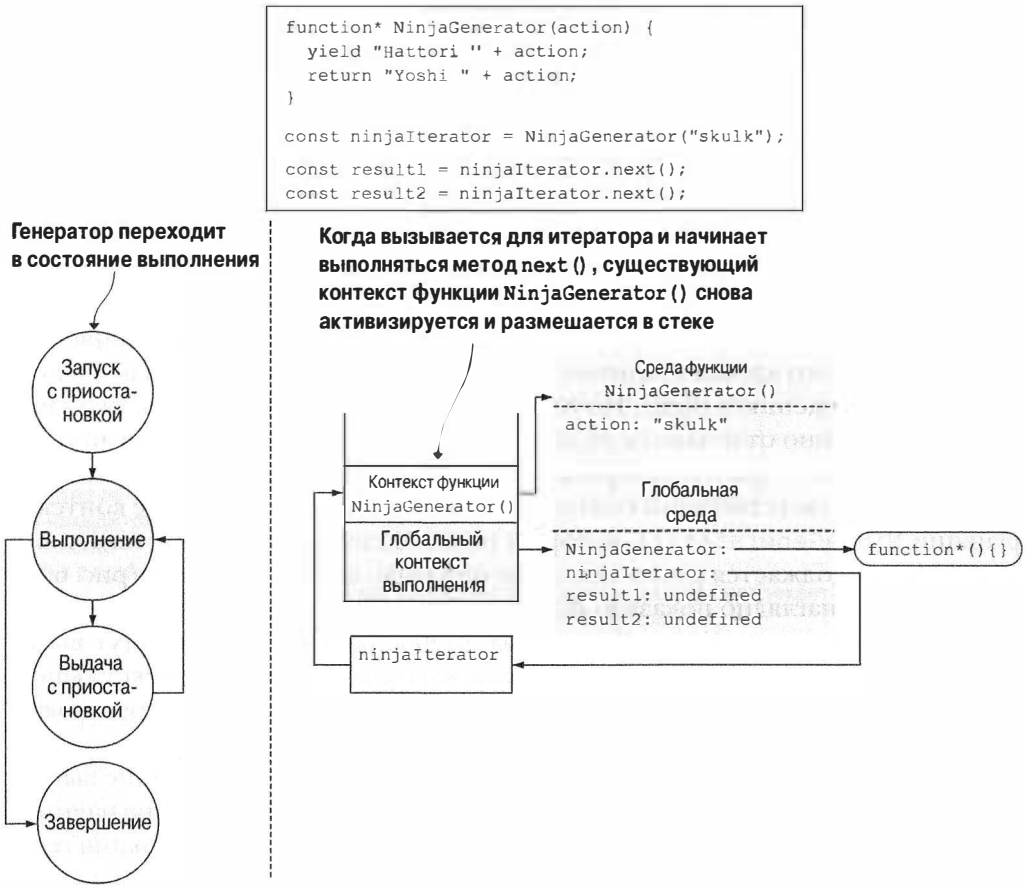


Рис. 6.8. Вызов метода **next()** итератора приводит к повторной активизации контекста выполнения соответствующего генератора, его размещению в стеке и продолжению выполнения с того места, где оно было остановлено в прошлый раз

Выполнение прикладного кода продолжается и достигает очередного вызова метода `next()` для итератора:

```
const result2 = ninjaIterator.next();
```

В этот момент весь процесс повторяется снова: контекст функции `NinjaGenerator()`, доступный по ссылке, хранящейся в объекте `ninjaIterator`, снова активизируется и размещается в стеке, а выполнение продолжается с того места, где оно было приостановлено. В данном случае генератор вычисляет выражение `"Yoshi " + action`. Но на этот раз вместо оператора `yield` в программе встречается оператор `return`, поэтому возвращается строковое значение `"Yoshi skulk"`, а генератор завершает свое выполнение, переходя в состояние завершения.

```
function* NinjaGenerator(action) {
  yield "Hattori " + action;
  return "Yoshi " + action;
}

const ninjaIterator = NinjaGenerator("skulk");

const result1 = ninjaIterator.next();
const result2 = ninjaIterator.next();
```

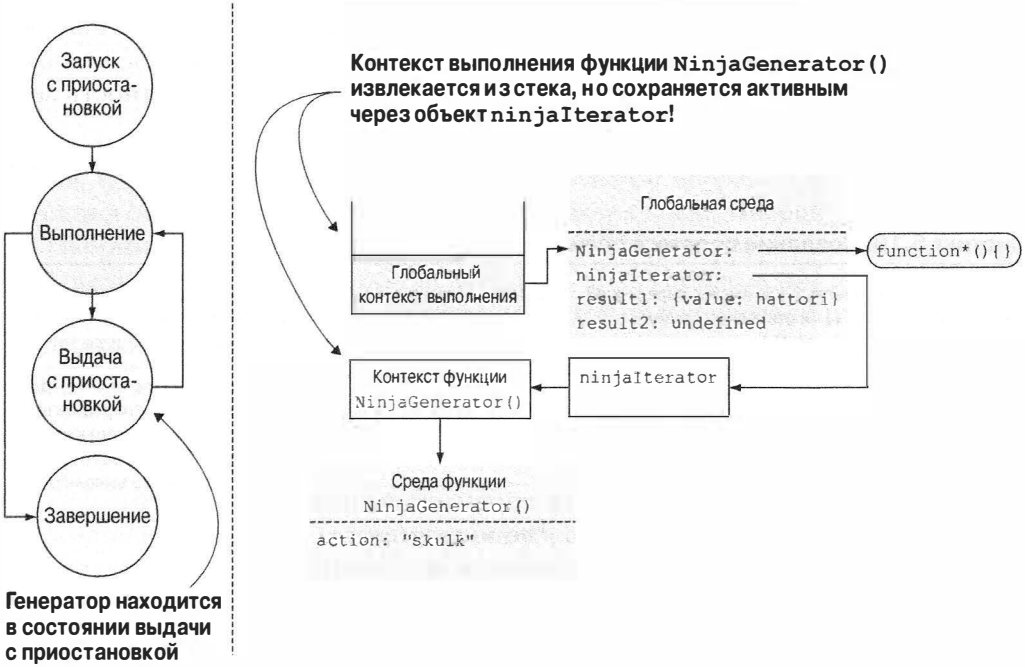


Рис. 6.9. После выдачи значения контекст выполнения генератора извлекается из стека, но не теряется, поскольку ссылка на него сохраняется в объекте `ninjaIterator`. А выполнение генератора приостанавливается, и он переходит в состояние выдачи с приостановкой

Итак, глубоко проникнув во внутренний механизм действия генераторов, мы показали, что все примечательные преимущества генераторов вытекают как побочный эффект из того обстоятельства, что контекст выполнения генератора поддерживается активным после возврата из него и не разрушается, как это обычно происходит после возврата значений из стандартных функций.

А теперь рекомендуем сделать небольшой перерыв, прежде чем перейти к рассмотрению обещаний — второй важной составляющей, необходимой для написания изящного асинхронного кода.

6.3. Работа с обещаниями

В языке JavaScript приходится в значительной степени задействовать асинхронные вычисления, результаты которых получаются не сразу, а в какой-то

определенный момент впоследствии. В связи с этим в стандарт ES6 было внедрено новое понятие обещаний, призванное упростить решение асинхронных задач.

Обещание — это заполнитель значения, которое отсутствует в настоящий момент, но появится впоследствии. Оно гарантирует, что в конечном итоге результат асинхронного вычисления станет известным. Если выполнить обещание, то в конечном итоге будет получено значение. А если возникнет затруднение, то в результате будет получена ошибка в качестве извинения за невозможность предоставить обещанное значение. Характерным примером применения обещаний служит получение данных от сервера. В этом случае обещается, что данные в конечном итоге будут получены, но всегда существует вероятность возникновения затруднений.

Создать новое обещание нетрудно, как демонстрируется в примере кода из листинга 6.10.

Листинг 6.10. Создание простого обещания

```
const ninjaPromise = new Promise((resolve, reject) => {
  resolve("Hattori");
  // reject("An error resolving a promise!");
});

ninjaPromise.then(ninja => {
  assert(ninja === "Hattori", "We were promised Hattori!");
}, err => {
  fail("There shouldn't be an error")
});
```

Обещание выполняется путем вызова переданной функции `resolve()` (и отклоняется путем вызова функции `reject()`)

Создать обещание, вызвав встроенный конструктор объектов типа `Promise` и передать ему функцию обратного вызова с двумя параметрами, `resolve` и `reject`

Вызывая метод `then()` для обещания, можно передать ему две функции обратного вызова. Первая из них вызывается, если обещание выполняется

А вторая функция вызывается, если возникает ошибка

Чтобы создать обещание, следует воспользоваться операцией `new` и встроенным конструктором объектов типа `Promise`, которому передается функция (в данном случае стрелочная функция, хотя вместо нее нетрудно указать и функциональное выражение). Эта функция называется *исполнителем* и имеет два параметра: `resolve` и `reject`. Исполнитель *немедленно* вызывается при конструировании объекта типа `Promise` с двумя встроенными функциями в качестве аргументов. В частности, функция `resolve()` вызывается вручную, если требуется выполнить обещание, а функция `reject()` — если возникает ошибка.

В рассматриваемом здесь примере кода применяется обещание, и с этой целью вызывается встроенный метод `then()` для объекта типа `Promise`. Этому методу передаются две функции обратного вызова. Первая функция вызывается при выполнении обещания, т.е. при вызове функции `resolve()` для обещания, а вторая — при отклонении обещания, т.е. при возникновении необрабатываемого исключении или вызове функции `reject()` для обещания.

В рассматриваемом здесь примере кода обещание создается и немедленно выполняется путем вызова функции `resolve()` с аргументом `Hattori`.

Следовательно, когда вызывается метод `then()`, при выполнении обещания запускается первая функция обратного вызова, где проверяется утверждение, и если оно проходит, то выводится сообщение `"We were promised Hattori!"` (Нам обещали Хаттори!).

Итак, пояснив принцип действия обещаний в общих чертах, перейдем к рассмотрению тех затруднений, разрешить которые они и были призваны.

6.3.1. Затруднения, связанные с простыми обратными вызовами

Асинхронный код применяется потому, что необходимо каким-то образом исключить блокирование работы приложения, которое может вызвать разочарование пользователей в ходе выполнения длительных операций. До настоящего времени подобное затруднение разрешалось с помощью обратных вызовов. С этой целью для выполнения длительной операции предоставляется функция обратного вызова, которая вызывается после завершения задания.

Например, получение файла в формате JSON от сервера является длительной операцией, в ходе которой желательно, чтобы приложение не переставало реагировать на действия пользователей. С этой целью предоставляется следующая функция обратного вызова, которая будет вызвана, как только завершится задание:

```
getJSON("data/ninjas.json", function() {
  /* обработать результаты */
});
```

Естественно, что в ходе выполнения длительной операции могут возникнуть ошибки. А трудность применения обратных вызовов связана с невозможностью использовать встроенные языковые конструкции вроде блоков операторов `try/catch` следующим образом:

```
try {
  getJSON("data/ninjas.json", function() {
    // обработать результаты
  });
} catch(e) {/* обработать ошибки */}
```

Это происходит потому, что код, в котором делается обратный вызов, обычно не выполняется на том же самом этапе цикла ожидания событий, где начинается длительная операция. (Вам станет ясно, что здесь имеется в виду, когда вы узнаете больше о цикле ожидания событий из материала главы 13.)

Таким образом, ошибки, как правило, теряются. Именно поэтому во многих библиотеках JavaScript приняты собственные соглашения для сообщения об ошибках. Например, на платформе Node.js функциям обратного вызова обычно передается два аргумента, `err` и `data`, где `err` принимает непустое значение, если по ходу дела в каком-нибудь другом месте кода возникает ошибка. И это приводит к первому затруднению, связанному с обратными вызовами: *трудностям обработки ошибок*.

После выполнения длительной операции нередко требуется каким-то образом обработать полученные данные. А это может привести к запуску на вы-

полнение другой длительной операции, которая, в свою очередь, запустит еще одну длительную операцию, и т.д., после чего последует ряд независимых, асинхронных шагов, требующих обработки при обратных вызовах. Так, если требуется реализовать хитрый план задействовать всех имеющихся ниндзя, найти место для первого ниндзя и отдать ему распоряжения, это намерение в конечном итоге приведет к следующему коду:

```
getJSON("data/ninjas.json", function(err, ninjas){
  getJSON(ninjas[0].location, function(err, locationInfo){
    sendOrder(locationInfo, function(err, status){
      /* обработать состояние */
    })
  })
});
```

Вам, вероятно, приходилось хотя бы раз иметь дело с аналогично структурированным кодом, где имеется целый ряд вложенных обратных вызовов, представляющих последовательность шагов, которые требуется предпринять. Согласитесь, что понять такой код нелегко, ввести в него новые шаги непросто, а организовать обработку ошибок еще труднее. Такая гора трудностей только нарастает по мере усложнения решаемых задач. И это приводит ко второму затруднению, связанному с обратными вызовами: *трудностям выполнения последовательности шагов*.

Иногда шаги, которые требуется предпринять, чтобы достичь конечного результата, не зависят друг от друга, и поэтому их лучше выполнять не в определенной последовательности, а параллельно ради экономии драгоценных миллисекунд. Так, если требуется составить план действий, исходя из имеющихся ниндзя, то для самого плана и того места, где его предполагается реализовать, можно выгодно воспользоваться методом `get()` из библиотеки jQuery, написав код, аналогичный следующему:

```
var ninjas, mapInfo, plan;

$.get("data/ninjas.json", function(err, data){
  if(err) { processError(err); return; }
  ninjas = data;
  actionItemArrived();
});

$.get("data/mapInfo.json", function(err, data){
  if(err) { processError(err); return; }
  mapInfo = data;
  actionItemArrived();
});

$.get("plan.json", function(err, data) {
  if(err) { processError(err); return; }

  plan = data;
  actionItemArrived ();
});
```

```
function actionItemArrived(){
  if(ninjas != null && mapInfo != null && plan != null){
    console.log("The plan is ready to be set in motion!");
  }
}

function processError(err){
  alert("Error", err)
}
```

В данном коде действия, направленные на получение сведений о ниндзя и карте местности, а также составление плана, выполняются параллельно, поскольку они не зависят друг от друга. И в конечном счете нужно получить в свое распоряжение все необходимые данные. А поскольку заранее неизвестно, в каком именно порядке получаются эти данные, то, получая их, приходится всякий раз проверять, являются ли они последней недостающей частью общего плана. И, наконец, когда все части плана окажутся на своих местах, можно привести план в действие. Следует, однако, иметь в виду, что для обычной организации параллельного выполнения ряда действий придется написать немало шаблонного кода. И это приводит к третьему затруднению, связанному с обратными вызовами: *трудностям параллельного выполнения ряда шагов*.

Представляя первое затруднение, связанное с обратными вызовами (т.е. трудности обработки ошибок), мы указали на отсутствие возможности пользоваться основополагающими конструкциями вроде блока операторов `try/catch`. Это же относится и к циклам. Так, если требуется выполнить асинхронные действия над каждым элементом коллекции, то придется преодолеть дополнительные трудности, чтобы достичь желанной цели.

Можно, конечно, создать библиотеку, упрощающую разрешение всех упомянутых выше затруднений, и многие так и поступают. Но зачастую это приводит к тому, что для разрешения одинаковых затруднений применяются разные подходы, и поэтому создатели языка JavaScript одарили разработчиков *обещаниями* в качестве стандартного подхода к организации асинхронных вычислений.

Теперь, когда стали ясны основные причины для внедрения обещаний и основное их назначение, рассмотрим их более подробно.

6.3.2. Углубленное исследование обещаний

Обещание — это объект, который служит в качестве заполнителя результата выполнения асинхронного задания. Оно представляет значение, которого пока еще нет, но есть надежда, что оно появится в будущем. Именно поэтому обещание может пройти в течение всего срока своего действия ряд состояний, как показано на рис. 6.10.

Сразу после создания обещание находится в состоянии *ожидания разрешения*, в котором ничего неизвестно об обещанном значении. Именно поэтому обещание в состоянии ожидания разрешения называется иначе *неразрешенным обещанием*. Если в ходе выполнения программы вызывается функция `resolve()`

данного обещания, оно переходит в состояние "выполнено", в котором успешно получено обещанное значение. А если вызывается функция `reject()` или возникает необрабатываемое исключение во время обработки данного обещания, то оно переходит в состояние "отклонено", в котором нельзя получить обещанное значение, но, по крайней мере, известна причина этого. Как только обещание достигнет состояния "выполнено" или "отклонено", оно так и остается в этом состоянии, т.е. обещание не может изменить состояние "выполнено" на состояние "отклонено", и наоборот. Таким образом, можно сказать, что обещание *разрешено*, будь то удачно или неудачно.

Рис. 6.10. Состояния обещания

В примере кода из листинга 6.11 наглядно показано, что происходит, когда применяются обещания.

Листинг 6.11. Порядок выполнения обещаний

```
report("At code start");

var ninjaDelayedPromise = new Promise((resolve, reject) => {
  report("ninjaDelayedPromise executor");
  setTimeout(() => {
    report("Resolving ninjaDelayedPromise");
    resolve("Hattori");
  }, 500);
});

assert(ninjaDelayedPromise !== null,
       "After creating ninjaDelayedPromise");

ninjaDelayedPromise.then(ninja => {
  assert(ninja === "Hattori",
         "ninjaDelayedPromise resolve handled with Hattori");
});
```

Вызов конструктора объектов типа `Promise` немедленно приводит к вызову переданной ему функции

Это обещание предполагается удачно разрешить по истечении времени ожидания 500 мс

Метод `then()` объекта типа `Promise` служит для установки функции обратного вызова, которая вызывается, когда разрешается обещание (в данном случае истекает время ожидания)

```
const ninjaImmediatePromise = new Promise((resolve, reject) => {
  report("ninjaImmediatePromise executor. Immediate resolve.");
  resolve("Yoshi");
});

ninjaImmediatePromise.then(ninja => {
  assert(ninja === "Yoshi",
         "ninjaImmediatePromise resolve handled with Yoshi");
});

report("At code end");
```

Создать новое обещание, которое немедленно разрешается

Установить функцию обратного вызова, которая вызывается после разрешения обещания. Но ведь оно уже разрешено!

При выполнения кода из листинга 6.11 выводятся результаты, приведенные на рис. 6.11. Как видите, этот код начинается с записи в системный журнал, т.е. вывода сообщения `"At code start"` (Вначале выполнения кода) с помощью специальной функции `report()`, которая выводит это сообщение на экран (подробнее о ведении системного журнала см. в приложении Б). Это дает возможность без особого труда отслеживать порядок выполнения кода.

Далее в рассматриваемом здесь коде создается новое обещание, для чего вызывается конструктор объектов типа `Promise`. Это немедленно приводит к вызову функции-исполнителя, где время ожидания задается следующим образом:

```
setTimeout(() => {
  report("Resolving ninjaDelayedPromise");
  resolve("Hattori");
}, 500);
```

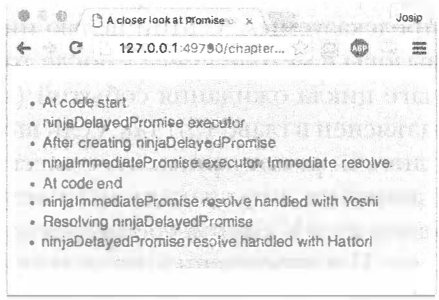


Рис. 6.11. Результат выполнения кода из листинга 6.11

Обещание разрешается по истечении установленного времени ожидания 500 мс. И хотя это могла быть любая другая асинхронная операция, тем не менее, в данном примере кода ради простоты была выбрана скромная задержка во времени.

После создания объекта обещания `ninjaDelayedPromise` еще неизвестно значение, которое в конечном итоге появится, и будет ли это вообще осуществлено удачно. (Напомним, что обещание все еще ожидает своего разрешения в течение заданного интервала времени.) Следовательно, после своего созда-

ния объект обещания ninjaDelayedPromise находится в своем первоначальном состоянии *ожидания разрешения*.

Далее для объекта обещания ninjaDelayedPromise вызывается метод then(), чтобы запланировать обратный вызов, который будет сделан, как только обещание успешно разрешится, как показано ниже. Такой обратный вызов *всегда* делается асинхронно независимо от текущего состояния обещания.

```
ninjaDelayedPromise.then(ninja => {
  assert(ninja === "Hattori",
         "ninjaDelayedPromise resolve handled with Hattori");
});
```

После этого создается еще один объект обещания ninjaImmediatePromise, которое разрешается немедленно во время своего создания, для чего вызывается функция resolve(). В отличие от объекта обещания ninjaDelayedPromise, которое сразу же переходит в состояние *ожидания разрешения* после своего создания, создание объекта обещания ninjaImmediatePromise завершается в *разрешенном* состоянии, и обещание сразу же получает строковое значение "Yoshi".

Вслед за этим вызывается метод then() для объекта обещания ninja ImmediatePromise, чтобы зарегистрировать обратный вызов, который будет сделан, как только обещание удачно разрешится. Но если обещание уже выполнено, то будет ли сделан обратный вызов немедленно при удачном разрешении обещания или же он будет проигнорирован? Ни то и ни другое.

Обещания служат для выполнения асинхронных действий, и поэтому интерпретатор JavaScript всегда прибегает к асинхронной обработке, чтобы сделать поведение обещания предсказуемым. С этой целью интерпретатор JavaScript выполняет обратные вызовы в методе then() после того, как будет выполнен весь код на текущем шаге цикла ожидания событий (напомним, что этот вопрос будет подробно разъяснен в главе 13). Так, если внимательно проанализировать результат выполнения рассматриваемого здесь кода, представленный на рис. 6.11, то можно заметить, что сначала выводится сообщение "At code end" (В конце выполнения кода), а затем сообщение о разрешении обещания ninjaImmediatePromise. И в завершение, по истечении времени ожидания 500 мс, выполняется обещание ninjaDelayedPromise, что приводит к запуску соответствующего обратного вызова в методе then().

Ради простоты в данном примере был реализован лишь идеальный сценарий, когда все проходит нормально. Но в действительности не все так идеально, поэтому рассмотрим те осложнения, которые могут возникнуть в связи с обещаниями.

6.3.3. Отклонение обещаний

Отклонить обещание можно двумя способами: *явно*, вызвав переданную функцию reject() в теле функции-исполнителя обещания, или *неявно*, если во

время обработки обещания возникает необрабатываемое исключение. Начнем исследование способов отклонения обещаний с примера кода из листинга 6.12.

Листинг 6.12. Явное отклонение обещаний

```
const promise = new Promise((resolve, reject) => {
  reject("Explicitly reject a promise!");
});

promise.then(
  () => fail("Happy path, won't be called!"),
  error => pass("A promise was explicitly rejected!")
);
```

Обещание можно отклонить явно, вызвав переданную функцию `reject()`

Если обещание отклоняется, то делается второй обратный вызов, предоставляемый на случай ошибки

Обещание можно отклонить явно, вызвав функцию `reject("Explicitly reject a promise!")`. Если обещание отклоняется, то всегда делается второй обратный вызов, зарегистрированный в методе `then()` на случай ошибки.

Для обработки отклонений обещаний можно также воспользоваться альтернативным синтаксисом, вызвав по цепочке встроенный метод `catch()`, как демонстрируется в примере кода из листинга 6.13.

Листинг 6.13. Вызов метода `catch()` по цепочке

```
var promise = new Promise((resolve, reject) => {
  reject("Explicitly reject a promise!");
});
promise.then(()=> fail("Happy path, won't be called!"))
       .catch(() => pass("Promise was also rejected"));
```

Вместо того чтобы предоставлять второй обратный вызов на случай ошибки, можно вызвать по цепочке метод catch(). Ему передается функция обратного вызова, которая запускается при возникновении ошибки. В итоге вы добьетесь того же результата!

Как показано в примере кода из листинга 6.13, метод `catch()` можно вызвать по цепочке после метода `then()` с той же самой целью — предоставить функцию обратного вызова на случай ошибки. Она вызывается при отклонении обещания. Какой из вариантов кода предпочесть — это дело личных предпочтений. Оба рассмотренных здесь варианта равноценны, но второй служит наглядным примером того, насколько удобно связывать метод `catch()` в цепочку.

Помимо явного отклонения (путем вызова функции `reject()`), обещание можно отклонить и неявно, если во время его обработки возникнет исключение. Рассмотрим в качестве примера код из листинга 6.14.

Листинг 6.14. Неявное отклонение обещаний посредством исключений

```
const promise = new Promise((resolve, reject) => {
  undeclaredVariable++;
});
```

Обещание отклоняется неявно, если при обработке обещания возникает необрабатываемое исключение

```
promise.then(() => fail("Happy path, won't be called!"))
       .catch(error => pass("Third promise was also rejected"));
```

Если возникает исключение, то делается второй обратный вызов, предоставляемый на случай ошибки

В теле исполнителя обещания предпринимается попытка инкрементировать значение переменной `undeclaredVariable`, которая не определена в программе. Как и следовало ожидать, это приведет к исключению. Но поскольку в теле функции-исполнителя отсутствует блок операторов `try/catch`, текущее обещание отклоняется неявно и в конечном итоге делается обратный вызов, указанный в методе `catch()`. В подобной ситуации можно было бы просто предоставить на случай ошибки второй обратный вызов в методе `then()`, чтобы добиться того же самого результата.

Такой единообразный подход к разрешению затруднений, возникающих при обращении с обещаниями, очень удобен. Независимо от того, каким образом отклоняется обещание, будь то явно путем вызова функции `reject()` или неявно при возникновении исключения, все ошибки и причины для отклонения направляются функции обратного вызова при отклонении. И тем самым задача разработчиков немного упрощается.

Итак, выяснив, каким образом действуют обещания и как планируются обратные вызовы при выполнении и отклонении обещаний, обратимся к практическому примеру получения данных в формате JSON от сервера, "пообещав" их доставку.

6.3.4. Создание первого настоящего обещания

К числу самых распространенных действий, предпринимаемых на стороне клиента, относится получение данных от сервера. И это действие служит превосходным практическим примером применения обещаний. Для базовой реализации такого примера мы воспользуемся встроенным объектом `XMLHttpRequest`, как показано в коде из листинга 6.15.

Примечание

Для выполнения кода из данного и всех последующих примеров, в которых применяется функция **`getJSON()`**, потребуется действующий сервер. С этой целью можно, например, воспользоваться сервером, доступным по адресу **`https://www.npmjs.com/package/http-server`**.

Листинг 6.15. Создание обещания `getJSON`

Создать объект XMLHttpRequest

Создать и возвратить новое обещание

```
function getJSON(url) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
```

Инициализировать запрос

```
    request.open("GET", url);
```

В данном примере преследуется цель создать функцию `getJSON()`, возвращающую обещание, которое позволяет зарегистрировать обратные вызовы, запускаемые при удачном или неудачном исходе асинхронного получения данных в формате JSON от сервера. Для базовой реализации в данном примере используется встроенный объект `XMLHttpRequest`, предоставляющий два события: `onload` и `onerror`. В частности, событие `onload` инициируется, когда браузер получает ответ от сервера, а событие `onerror` — когда возникает ошибка в ходе обмена данными. Обработчики этих событий будут вызываться браузером асинхронно по мере их наступления.

Если в ходе обмена данными возникнет ошибка, то получать данные от сервера мы уже не сможем. Поэтому лучше всего отклонить обещание следующим образом:

```
request.onerror = function(){
  reject(this.status + " " + this.statusText);
};
```

Получив ответ от сервера, можно проанализировать его и рассмотреть конкретную ситуацию. Не вдаваясь особенно в подробности, следует сказать, что сервер может прислать в ответ различные данные, но в данном случае нас интересует только удачный ответ (код состояния **200**). В противном случае обещание отклоняется.

Даже если в ответ сервер без ошибок прислал какие-то данные, то это еще не означает, что все в порядке. А поскольку мы преследуем цель получить от сервера строку в формате JSON, то должны иметь в виду, что код JSON может всег-

да содержать синтаксические ошибки. Именно поэтому вызов метода `JSON.parse()` заключен в блок операторов `try/catch`. Если при синтаксическом анализе полученного ответа возникает исключение, то и в этом случае обещание отклоняется. Таким образом, мы учли все возможные неудачные сценарии.

Если все пойдет по плану и мы успешно получим свои данные, то сможем благополучно выполнить обещание. И, наконец, мы можем вызвать функцию `getJSON()`, чтобы получить сведения о ниндзя от сервера:

```
getJSON("data/ninjas.json").then(ninjas => {
  assert(ninjas !== null, "Ninjas obtained!");
}).catch(e => fail("Shouldn't be here:" + e));
```

В данном случае у нас имеются три потенциальных источника ошибок: ошибки при установлении соединения клиента с сервером, ошибки в связи с отправкой сервером непредвиденных данных в ответ (состояние недостоверного ответа), а также ошибки в связи с получением недостоверных данных в формате JSON. Но что касается прикладного кода, в котором применяется функция `getJSON()`, то нас не интересуют особенности источников ошибок. Мы только предоставляем обратный вызов, который запускается, если все прошло нормально и данные получены надлежащим образом, а также обратный вызов, который вызывается, если возникнет какая-нибудь ошибка. И благодаря этому задача разработчиков немного упрощается.

А теперь пойдем дальше, исследовав еще одно преимущество обещаний, которое заключается в изящности их составления. И начнем мы со связывания нескольких обещаний в цепочку последовательных вызовов.

6.3.5. Связывание обещаний в цепочку

Как было показано ранее, выполнение последовательности независимых шагов приводит к накапливанию целой горы трудностей, тесно связанных с трудностями поддержания последовательности обратных вызовов. Обещания являются шагами в направлении разрешения данного затруднения, поскольку их можно связывать в цепочку.

Ранее в этой главе было также показано, что с помощью метода `then()`, вызываемого для обещания, можно зарегистрировать обратный вызов, который будет сделан при удачном выполнении обещания. Но при этом не было показано, что в результате вызова метода `then()` также возвращается новое обещание. Следовательно, ничто не мешает нам связать в цепочку столько методов `then()`, сколько потребуется, как показано в примере кода из листинга 6.16.

В данном примере кода создается последовательность обещаний, которые будут разрешены по очереди, если все пойдет по плану. Сначала вызывается метод `getJSON("data/ninjas.json")`, чтобы извлечь список ниндзя из файла на сервере. После получения этого списка на основании сведений о первом ниндзя составляется первый запрос перечня заданий, порученных ниндзя: `getJSON(ninjas[0].missionsUrl)`. А затем, когда эти задания поступят, составляется еще один запрос подробностей первого запроса:

getJSON(missions[0].detailsUrl). И, наконец, выводятся подробности задания.

Листинг 6.16. Связывание обещаний в цепочку с помощью методов then()

```
getJSON("data/ninjas.json")
  .then(ninjas => getJSON(ninjas[0].missionsUrl))
  .then(missions => getJSON(missions[0].detailsUrl))
  .then(mission => assert(mission !== null, "Ninja mission obtained!"))
  .catch(error => fail("An error has occurred"));
```

Перехватить отклонения обещаний на любом шаге

Указать ряд последовательных шагов, связав в цепочку вызовы метода then()

Написание такого кода с помощью стандартных обратных вызовов привело бы к глубокому вложению обратных вызовов, а выявить конкретную последовательность шагов было бы непросто. И не дай Бог, еще решиться на ввод дополнительного шага где-то посредине!

Перехват ошибок в обещаниях, связанных в цепочку

При выполнении последовательностей асинхронных шагов на любом из них может возникнуть ошибка. Как пояснялось ранее, при вызове метода then() можно указать второй обратный вызов на случай ошибки или вызвать по цепочке метод catch(), которому передается функция обратного вызова на случай ошибки. Если же речь идет об удачном или неудачном завершении отдельных шагов, то снабдить каждый такой шаг отдельным обработчиком ошибок было бы затруднительно. Поэтому можно воспользоваться методом catch(), как было показано ранее в листинге 6.16 и еще раз демонстрируется ниже.

```
...catch(err => fail("An error has occurred:" + err));
```

Если произойдет ошибка в любом из предыдущих обещаний, она будет перехвачена в методе catch(). А если ошибки не возникнут, то выполнение программы беспрепятственно продолжится дальше.

Согласитесь, что выполнять последовательность шагов намного проще с помощью обещаний, а не обычных обратных вызовов. Но и такой способ все еще нельзя назвать изящным. Мы еще вернемся к этому вопросу, а до тех пор покажем, как организовать параллельное выполнение асинхронных шагов с помощью обещаний.

6.3.6. Ожидание ряда обещаний

Помимо возможности выполнять последовательности независимых асинхронных шагов, обещания существенно сокращают бремя ожидания завершения нескольких независимо выполняемых асинхронных заданий. Вернемся к примеру, в котором требовалось параллельно собрать сведения об имеющихся ниндзя, подробностях плана действий и карты местности, где предполагается

реализовать этот план. Обещания позволяют сделать это довольно просто, как показано в примере кода из листинга 6.17.

Как видите, не имеет никакого значения порядок выполнения заданий и завершены ли одни из них прежде других. В данном примере кода вызывается метод `Promise.all()` в ожидании ряда обещаний. Этому методу передается массив обещаний. Он создает *новое* обещание, которое выполняется в том случае, если выполняются все переданные обещания, или отклоняется, если будет отклонено хотя бы одно из этих обещаний. Функции обратного вызова, запускаемой при удачном исходе, передается массив значений, удачно разрешенных в порядке передачи обещаний. Проанализируйте код из данного примера, оценив изящество параллельной обработки в нем нескольких асинхронных операций с помощью обещаний.

Листинг 6.17. Ожидание ряда обещаний с помощью метода `Promise.all()`

```
Promise.all([getJSON("data/ninjas.json"),
             getJSON("data/mapInfo.json"),
             getJSON("data/plan.json")]).then(results => {
               const ninjas = results[0], mapInfo = results[1],
                             plan = results[2];
               assert(ninjas !== undefined
                      && mapInfo !== undefined && plan !== undefined,
                      "The plan is ready to be set in motion!");
}).catch(error => {
  fail("A problem in carrying out our plan!");
});
```

В результате получается массив удачно разрешенных значений, располагаемых в порядке переданных обещаний

Методу `Promise.all()` передается массив обещаний. Он создает новое обещание, которое выполняется, если удачно выполняются все остальные обещания, и отклоняется, если отклоняется хотя бы одно из обещаний

Метод `Promise.all()` ожидает все обещания в списке. Но иногда из целого ряда обещаний наибольший интерес представляет лишь первое обещание, которое должно быть выполнено или отклонено. И для этой цели служит метод `Promise.race()`.

Состязание обещаний

Допустим, что в нашем распоряжении имеется группа ниндзя и нам требуется поручить задание первому же ниндзя, который откликнется на наш призыв. Эту задачу можно решить с помощью обещаний, написав код, аналогичный приведенному в листинге 6.18.

Листинг 6.18. Состязание обещаний с помощью метода `Promise.race()`

```
Promise.race([getJSON("data/yoshi.json"),
              getJSON("data/hattori.json"),
              getJSON("data/hanzo.json")])
    .then(ninja => {
       assert(ninja !== null, ninja.name + " responded first");
     }).catch(error => fail("Failure!"));
```

Что может быть проще, чем этот код, где ничего не нужно отслеживать вручную. Когда вызывается метод `Promise.race()`, ему передается массив обещаний, а он возвращает совершенно *новое* обещание, которое выполняется или отклоняется, как только будет выполнено или отклонено первое же обещание.

В предыдущих примерах было показано, каким образом действуют обещания и как с их помощью значительно упрощается выполнение (как последовательно, так и параллельно) ряда асинхронных шагов. Но, несмотря на все усовершенствования с точки зрения обработки ошибок и изящества по сравнению с обыкновенными обратными вызовами, код, наделенный обещаниями, не настолько прост и изящен, как синхронный. Поэтому в следующем разделе два главных понятия *генераторов* и *обещаний*, представленных в этой главе, применяются вместе. Это позволяет достичь в асинхронном коде простоты синхронного кода, которому присущ неблокирующий характер.

6.4. Сочетание генераторов и обещаний

В этом разделе поясняется, как сочетать генераторы (и их способность приостанавливать и возобновлять свою работу) с обещаниями, чтобы достичь большего изящества асинхронного кода. С этой целью мы обратимся к примеру функциональных возможностей, благодаря которым пользователи могут получать подробные сведения о наиболее трудных заданиях, выполненных самыми известными нинзя. Сведения о нинзя, сводка выполненных ими заданий, а также подробные сведения о самих заданиях хранятся в формате JSON на удаленном сервере.

Все эти подзадачи носят длительный характер и взаимозависимы. Если реализовать их синхронным способом, то в конечном итоге получится следующий простой код:

```
try {
  const ninjas = syncGetJSON("data/ninjas.json");
  const missions = syncGetJSON(ninjas[0].missionsUrl);
  const missionDetails = syncGetJSON(missions[0].detailsUrl);
  // изучить описание задания
} catch(e){
  // получить подробности задания не удалось
}
```

Несмотря на всю простоту и изящество обработки ошибок в этом коде, он все же блокирует пользовательский интерфейс, что вряд ли удовлетворит пользователей. В идеальном случае этот код можно было бы видоизменить таким образом, чтобы исключить блокирование во время выполнения длительных операций. И это можно, в частности, сделать, сочетая генераторы с обещаниями.

Как известно, выдача значения из генератора приостанавливает его выполнение без блокировки. Чтобы возобновить выполнение генератора, придется вызвать метод `next()` для итератора этого генератора. С другой стороны, обе-

щания позволяют указать функции обратного вызова, которые запускаются как при успешном выполнении обещания, так и в случае возникновения ошибки.

В таком случае можно сочетать генераторы с обещаниями, разместив код для выполнения асинхронных операций в генераторе и запустив эту функцию-генератор. Как только в процессе выполнения генератора будет достигнут момент, когда вызывается асинхронная операция, создается обещание, представляющее значение этой операции. А поскольку заранее неизвестно, когда именно выполнится обещание (и выполнится ли оно вообще), то в данный момент мы покидаем генератор, чтобы исключить блокировку. Некоторое время спустя, когда обещание будет выполнено или отклонено, для итератора вызывается метод `next()`, чтобы продолжить выполнение генератора. И это делается столько раз, сколько потребуется. Практический пример реализации такого подхода приведен в коде из листинга 6.19.

Функция `async()` вызывает генератор и создает итератор, предназначенный для возобновления выполнения генератора. В теле функции `async()` объявляется функция `handle()`, обрабатывающая одно значение, возвращаемое из генератора (т.е. выполняется один шаг итератора). Если в результате работы генератора возвращается обещание, которое выполняется, то для итератора вызывается метод `next()`, чтобы отправить обещанное значение обратно генератору и возобновить работу генератора. А если возникнет ошибка и обещание будет отвергнуто, то эта ошибка генерируется для генератора с помощью метода `throw()` (как видите, он снова пригодился). Этот процесс продолжается до тех пор, пока генератор не сообщит о завершении своего выполнения.

Листинг 6.19. Сочетание генераторов с обещаниями

Определить вспомогательную функцию для управления генератором

Функция, использующая асинхронные результаты, должна быть в состоянии приостановить свое выполнение, ожидая эти результаты. Обратите внимание на обозначение `function*`, указывающее на применение генератора!

```
async(function*(){
  try {
    const ninjas = yield getJSON("data/ninjas.json");
    const missions = yield getJSON(ninjas[0].missionsUrl);
    const missionDescription = yield getJSON(missions[0].detailsUrl);
    // изучить описание задания
  }
  catch(e) {
    // получить подробности задания не удалось
  }
});

function async(generator) {
  var iterator = generator();

  function handle(iteratorResult) {
    if(iteratorResult.done) { return; }
```

Выдать результат выполнения каждой асинхронной операции

Можно по-прежнему пользоваться всеми стандартными языковыми конструкциями, в том числе блоками операторов `try/catch` и циклами

Создать итератор для управления генератором

Определить функцию для обработки каждого значения, сформированного генератором

Завершить работу, когда у генератора больше не останется результатов

```
    const iteratorValue = iteratorResult.value;

    if(iteratorValue instanceof Promise) {
      iteratorValue.then(res => handle(iterator.next(res)))
                   .catch(err => iterator.throw(err));
    }
  }
  try {
    handle(iterator.next());
  }
  catch (e) { iterator.throw(e); }
}
```

Если сгенерированное значение оказывается обещанием, зарегистрировать функции обратного вызова для обработки удачного и неудачного исхода. Это асинхронная часть кода. Если обещание выполняется, работа генератора возобновляется и от него получается обещанное значение. А если возникает ошибка, генерируется исключение для генератора

Возобновить выполнение генератора

Примечание

Данный пример служит лишь черновым вариантом с минимальным объемом кода, требующегося для сочетания генераторов с обещаниями и не рекомендуется для применения в практике разработки веб-приложений.

А теперь рассмотрим генератор из данного примера более подробно. При первом вызове метода `next()` для итератора выполнение генератора происходит вплоть до первого вызова `getJSON("data/ninjas.json")`. В результате этого вызова создается обещание, которое в конечном итоге содержит список сведений о ниндзя. А поскольку это значение извлекается асинхронно, то заранее неизвестно, сколько времени потребуется браузеру, чтобы получить его. Но известно другое: выполнение прикладной программы не должно быть заблокировано на время ожидания. Поэтому в данный момент генератор выдает в качестве своего значения обещание, приостанавливается и возвращает управление вызывающей функции `handle()`. А поскольку произведенное значение оказывается обещанием функции `getJSON()`, то с помощью методов `then()` и `catch()` объекта-обещания в функции `handle()` регистрируются обратные вызовы для обработки удачного и неудачного исхода, после чего выполнение продолжается. Далее управление переходит из функции `handle()` в тело функции `async()` (в данном случае код исчерпан, поэтому наступает бездействие). В это время функция-генератор находится в приостановленном состоянии и терпеливо ожидает события, не блокируя выполнение программы.

Гораздо позднее, когда браузер получает (положительный или отрицательный) ответ, делается один из обратных вызовов обещания. Если обещание выполнено, запускается функция обратного вызова для удачного исхода, что, в свою очередь, приводит к выполнению метода `next()` объекта-итератора, где у генератора запрашивается очередное значение. Генератор возобновляет свое выполнение, получая значение, передаваемое при обратном вызове. Это означает повторный вход в тело генератора после первого выражения `yield`, значением которого становится список ниндзя, который асинхронно запраши-

вается у сервера. Выполнение функции-генератора продолжается, и значение присваивается переменной `ninjas`.

В следующей строке кода генератора некоторые из полученных данных (`ninjas[0].missionUrl`) используются для очередного вызова функции `getJSON()`, где создается еще одно обещание, которое в конечном итоге должно содержать список заданий, выполненных самым известным ниндзя. А поскольку и это задание оказывается асинхронным, то заранее неизвестно, сколько времени потребуется для его выполнения. Поэтому выполнение генератора приостанавливается и весь процесс повторяется снова.

Данный процесс повторяется до тех пор, пока в генераторе имеются асинхронные задания. Рассмотренный здесь пример оказался более сложным, чем предыдущие, но он примечателен тем, что наглядно демонстрирует многое из того, что вы уже знаете, в том числе следующее.

- **Функции являются объектами высшего порядка.** Одну функцию можно передавать в качестве аргумента другой функции (в данном случае — `async()`).
- **Функции-генераторы.** В данном примере используется их способность приостанавливать и возобновлять выполнение.
- **Обещания.** Они помогают в работе с асинхронным кодом.
- **Обратные вызовы.** В обещаниях можно регистрировать функции обратного вызова, запускаемые при выполнении или отклонении обещаний.
- **Стрелочные функции.** Для обратных вызовов применяются стрелочные функции в силу их простоты.
- **Замыкания.** Итератор, через который осуществляется управление генератором, создается в функции `async()` и становится доступным через замыкания в функциях обратных вызовов обещаний.

Итак, рассмотрев весь упомянутый выше процесс, оценим, насколько более изящным оказывается код, реализующий исходную логику нашего приложения. В частности, рассмотрим следующий фрагмент кода:

```
getJSON("data/ninjas.json", (err, ninjas) => {
  if(err) {
    console.log("Error fetching ninjas", err);
    return;
  }

  getJSON(ninjas[0].missionsUrl, (err, missions) => {
    if(err) {
      console.log("Error locating ninja missions", err);
      return;
    }
    console.log(misssions);
  })
});
```

Вместо смеси операторов управления потоком выполнения программы и обработки ошибок и несколько сбивающего с толку кода в конечном счете получается нечто следующее:

```
async(function*() {
  try {
    const ninjas = yield getJSON("data/ninjas.json");
    const missions = yield getJSON(ninjas[0].missionsUrl);

    // вся полученная информация
  }
  catch(e) {
    // произошла ошибка
  }
});
```

В полученном конечном результате сочетаются преимущества синхронного и асинхронного кода. Из синхронного кода взято преимущество простоты для понимания и возможность пользоваться всеми стандартными механизмами управления потоком выполнения программы и обработки ошибок, в том числе блоками операторов `try/catch`. А из асинхронного кода взято преимущество неблокирующего характера, благодаря которому выполнение прикладного кода не блокируется в ожидании завершения длительной асинхронной операции.

6.4.1. Асинхронные функции в перспективе

Обратите внимание на то, что нам по-прежнему приходится писать шаблонный код и разрабатывать функцию `async()`, обрабатывающую обещания и запрашивающую значения у генератора. И хотя эту функцию достаточно написать лишь один раз и затем пользоваться ею неоднократно в прикладном коде, было бы еще лучше, если бы нам вообще не нужно было об этом думать. Создатели JavaScript вполне осознают пользу, которую приносит сочетание генераторов и обещаний, и поэтому они стремятся упростить задачу разработчиков, встраивая в язык непосредственную поддержку такого сочетания.

Для подобных случаев планируется внедрить в JavaScript два новых ключевых слова, `async` и `await`, исключающие потребность писать шаблонный код. В недалекой перспективе мы сможем писать код, аналогичный следующему:

```
(async function (){
  try {
    const ninjas = await getJSON("data/ninjas.json");
    const missions = await getJSON(missions[0].missionsUrl);
    console.log(missions);
  }
  catch(e) {
    console.log("Error: ", e);
  }
})()
```

Ключевое слово `async`, употребляемое перед ключевым словом `function`, специально указывает на то, что в данной функции используются асинхронные значения, и везде в прикладном коде, где вызывается асинхронное задание, размещается ключевое слово `await`, указывающее интерпретатору JavaScript на то, что результата следует ожидать без блокировки. В фоновом режиме происходит все, что обсуждалось ранее в этой главе, но теперь нам не придется об этом вообще беспокоиться.

Примечание

Асинхронные функции появятся в следующем выпуске JavaScript. В настоящее время они не поддерживаются в браузерах, но если вы желаете воспользоваться асинхронными функциями в своем коде, можете сделать это с помощью таких транспиляторов, как Babel или Traceur.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Генераторы являются функциями, генерирующими последовательности значений, но не все сразу, а по запросу.
- В отличие от стандартных функций, генераторы могут приостанавливать и возобновлять свое выполнение. Как только генератор сформирует значение, он приостановит свое выполнение, не блокируя основной поток исполнения и терпеливо ожидая следующего запроса.
- Генератор объявляется указанием знака звездочки (`*`) после ключевого слова `function`. В теле генератора используется новое ключевое слово `yield`, возвращающее значение и приостанавливающее выполнение генератора. Если же требуется запросить значение у другого генератора, то в теле генератора указывается оператор `yield*`.
- В результате вызова генератора создается объект-итератор, через который можно управлять процессом выполнения генератора. Значения запрашиваются у генератора с помощью метода `next()`, вызываемого для итератора, а вызывая метод `throw()` того же самого итератора, можно даже генерировать исключение для генератора. С помощью метода `next()` можно также посылать значения генератору.
- Обещание служит заполнителем результатов вычислений. Оно гарантирует, что результат вычислений (главным образом, асинхронных) в конечном итоге станет известен. Обещание может быть выполнено или отклонено, и после этого никаких изменений в его состоянии больше не предвидится.
- Обещания значительно упрощают работу с асинхронными операциями. В частности, последовательности независимых асинхронных шагов можно без труда выполнить, связывая обещания в цепочку с помощью метода

then(). Параллельное выполнение нескольких шагов можно также значительно упростить с помощью метода Promise.all().

- Генераторы и обещания можно сочетать вместе, чтобы работать с асинхронными операциями также просто, как и с синхронным кодом.

Упражнения

1. Какие значения примут переменные a1–a4 после выполнения приведенного ниже фрагмента кода?

```
function* EvenGenerator(){
  let num = 2;
  while(true){
    yield num;
    num = num + 2;
  }
}

let generator = EvenGenerator();

let a1 = generator.next().value;
let a2 = generator.next().value;
let a3 = EvenGenerator().next().value;
let a4 = generator.next().value;
```

2. Каким окажется содержимое массива ninjas после выполнения приведенного ниже фрагмента кода? (*Подсказка:* подумайте, как организовать цикл for-of с помощью цикла while.)

```
function* NinjaGenerator(){
  yield "Yoshi";
  return "Hattori";
  yield "Hanzo";
}

var ninjas = [];
for(let ninja of NinjaGenerator()){
  ninjas.push(ninja);
}

ninjas;
```

3. Какие значения примут переменные a1–a2 после выполнения приведенного ниже фрагмента кода?

```
function* Gen(val){
  val = yield val * 2;
  yield val;
}

let generator = Gen(2);
let a1 = generator.next(3).value;
let a2 = generator.next(5).value;
```

4. Каким окажется результат выполнения приведенного ниже фрагмента кода?

```
const promise = new Promise((resolve, reject) => {
  reject("Hattori");
});

promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

5. Каким окажется результат выполнения приведенного ниже фрагмента кода?

```
const promise = new Promise((resolve, reject) => {
  resolve("Hattori");
  setTimeout(()=> reject("Yoshi"), 500);
});

promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

Часть III

Исследование объектов и упрочение кода

Итак, рассмотрев особенности функций, продолжим наше исследование языка JavaScript, уделив пристальное внимание основам объектов в главе 7. О том, как управлять доступом к объектам и контролировать их состояние с помощью методов получения и установки, а также прокси-объектов, относящихся к совершенно новому типу объектов в JavaScript, речь пойдет в главе 8. А в главе 9 мы рассмотрим коллекции, в том числе традиционные массивы и такие совершенно новые типы коллекций, как отображения и множества.

После этого в главе 10 мы перейдем к рассмотрению регулярных выражений. Из этой главы вы узнаете, что задачи, для решения которых раньше требовалось немало кода, можно свести к дюжине операторов, если умело пользоваться регулярными выражениями в JavaScript.

И, наконец, в главе 11 будет показано, как структурировать приложения на JavaScript в более компактные, хорошо организованные функциональные единицы, называемые модулями.

7

Объектная ориентация с помощью прототипов

В этой главе...

- Исследование прототипов
- Применение функций в качестве конструкторов
- Расширение объектов с помощью прототипов
- Избежание типичных скрытых препятствий
- Построение классов средствами наследования

Мы выяснили, что функции являются объектами высшего порядка и оказываются необыкновенно универсальными и полезными в JavaScript благодаря замыканиям, а функции-генераторы можно эффективно сочетать с обещаниями для разрешения затруднений, связанных с асинхронным кодом. И теперь мы готовы перейти к рассмотрению прототипов объектов — еще одного важного языкового средства JavaScript.

Прототип — это объект, которому можно поручить поиск конкретного свойства. Прототипы являются удобными языковыми средствами для определения свойств и функциональных возможностей, которые автоматически становятся доступными для других объектов. Они служат той же цели, что и классы в традиционных объектно-ориентированных языках программирования. Безусловно, в JavaScript прототипы применяются, главным образом, для написания кода в объектно-ориентированном стиле, аналогично, хотя и не полностью соответствующего коду, написанному на более традиционных, основанных на классах языках вроде Java или C#.

В этой главе мы тщательно исследуем принцип действия прототипов, выясним их связь с функциями-конструкторами и покажем, как имитировать некоторые объектно-ориентированные средства, зачастую применяемые в других, более традиционных объектно-ориентированных языках программирования. Мы также рассмотрим новое для JavaScript ключевое слово `class`, которое позволяет без особого труда имитировать классы и наследование, хотя и не дает возможности реализовать полноценные классы в JavaScript. Итак, приступим к исследованию прототипов.

Знаете ли вы?

Как проверить, доступно ли объекту конкретное свойство?

Почему цепочка прототипов имеет особое значение для функционирования объектов в JavaScript?

Изменяют ли классы, введенные в стандарт ES6, функционирование объектов в JavaScript?

7.1. Общее представление о прототипах

Объекты в JavaScript являются коллекциями именованных свойств со значениями. Например, новые объекты можно без особого труда создать, воспользовавшись литеральной формой записи объекта следующим образом:

```
let obj = {
  prop1: 1,
  prop2: function(){},
  prop3: {}
}
```

Присвоить простое значение

Присвоить функцию

Присвоить другой объект

Как видите, свойствам объектов можно присваивать простые значения (например, числовые или строковые), функции и даже другие объекты. Кроме того, свойства, присваиваемые объекту, можно легко изменять и удалять благодаря высокой степени динамичности языка JavaScript, как показано ниже.

```
obj.prop1 = 1;
obj.prop1 = [];
delete obj.prop2;
```

В свойстве `prop1` сохраняется простое число

Присвоить значение совершенно другого типа (в данном случае — массив)

Удалить свойство из объекта

Объекты можно даже дополнять совершенно новыми свойствами:

```
obj.prop4 = "Hello";
```

Добавить совершенно новое свойство

В результате всех этих видоизменений рассматриваемый здесь простой объект останется в следующем состоянии:

```
{
  prop1: [],
  prop3: {},
  prop4: "Hello"
};
```

При разработке программного обеспечения следует стремиться к тому, чтобы не изобретать колесо и повторно использовать код как можно больше. Одной из форм повторного использования кода, помогающей также в организации прикладных программ, служит *наследование*, расширяющее возможности одного объекта в другом. В языке JavaScript наследование реализуется путем создания прототипов.

Создание прототипов опирается на простой принцип. У каждого объекта может быть ссылка на свой *прототип*, т.е. объект, в котором будет выполняться поиск конкретного свойства, если оно отсутствует у самого объекта. Представьте, что вы участвуете в викторине и ее ведущий задает вам вопрос. Если вы знаете ответ, то отвечаете сразу, а если не знаете, то обращаетесь за помощью к соседнему участнику викторины. Именно так и действуют прототипы.

В качестве примера рассмотрим код из листинга 7.1.

Листинг 7.1. С помощью прототипов одни объекты могут получать доступ к свойствам других объектов

```
const yoshi = { skulk: true };
const hattori = { sneak: true };
const kuma = { creep: true };

assert("skulk" in yoshi, "Yoshi can skulk");
assert(!("sneak" in yoshi)), "Yoshi cannot sneak");
assert(!("creep" in yoshi)), "Yoshi cannot creep");

Object.setPrototypeOf(yoshi, hattori);

assert("sneak" in yoshi, "Yoshi can now sneak");
assert(!("creep" in hattori)), "Hattori cannot creep");

Object.setPrototypeOf(hattori, kuma);

assert("creep" in hattori, "Hattori can now creep");
assert("creep" in yoshi, "Yoshi can also creep");
```

Создать три объекта, каждый со своим свойством

Объекту `yoshi` доступно только его собственное свойство `skulk`

Задать один объект в качестве прототипа другого объекта, вызвав метод `Object.setPrototypeOf()`

Если установить объект `hattori` в качестве прототипа объекта `yoshi`, то свойства объекта `hattori` станут доступными для объекта `yoshi`

Задать объект `kuma` в качестве прототипа объекта `hattori`

В настоящий момент свойство `creep` недоступно для объекта `hattori`

Теперь свойство `creep` доступно для объекта `hattori`

Свойство `creep` доступно и для объекта `yoshi` через прототип `hattori`

В начале данного примера кода создаются следующие три объекта: `yoshi`, `hattori` и `kuma`. И каждому из них доступно только одно свойство именно этого объекта. В частности, объекту `yoshi` доступно только свойство `skulk`, объекту `hattori` — только свойство `sneak`, а объекту `kuma` — только свойство `creep` (рис. 7.1).

Проверить, имеется ли у объекта доступ к конкретному свойству, можно с помощью операции `in`. Например, в результате выполнения операции `skulk in yoshi` возвращается логическое значение `true`, поскольку объекту `yoshi` доступно свойство `skulk`, тогда как в результате выполнения операции `sneak in yoshi` возвращается логическое значение `false`.

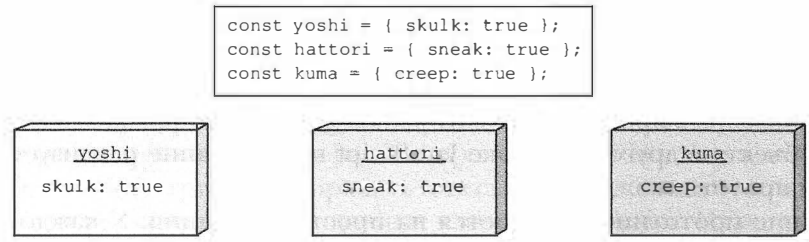


Рис. 7.1. Первоначально каждому объекту доступны только его собственные свойства

Ссылка на прототип в объекте JavaScript хранится в его внутреннем свойстве, к которому нельзя непосредственно получить доступ из кода, и поэтому оно обозначается как `[[prototype]]`. Поэтому для установки прототипов используется встроенный метод `Object.setPrototypeOf()`, которому в качестве аргументов передаются два объекта: второй из них будет служить прототипом первого. Так, если сделать следующий вызов:

```
Object.setPrototypeOf(yoshi, hattori);
```

то объект `hattori` будет задан в качестве прототипа объекта `yoshi`.

Всякий раз, когда запрашивается значение свойства, которое отсутствует у объекта `yoshi`, поиск этого свойства выполняется в объекте `hattori`. Таким образом, свойство `sneak` объекта `hattori` становится доступным через объект `yoshi` (рис. 7.2).

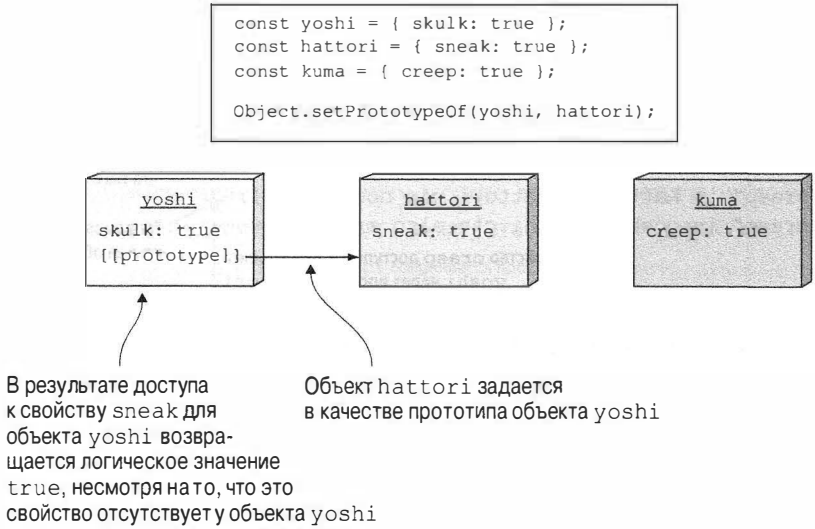


Рис. 7.2. Если требуется доступ к свойству, которое отсутствует у объекта, его поиск выполняется в прототипе данного объекта. В данном примере свойство **`sneak`** объекта **`hattori`** становится доступным через объект **`yoshi`**, поскольку первый из этих объектов является прототипом второго

То же самое можно сделать и с объектами hattori и kuma. В частности, с помощью метода Object.setPrototypeOf() можно задать объект kuma в качестве прототипа объекта hattori. Если затем запросить у объекта hattori свойство, которое у него отсутствует, он поручит искать его своему прототипу kuma. В данном случае свойство creep объекта kuma становится доступным объекту hattori (рис. 7.3).

Следует особо подчеркнуть, что прототип может быть у всякого объекта, а у прототипа объекта — свой прототип и т.д., в результате чего образуется *цепочка прототипов*. Поиск свойства выполняется по всей цепочке и прекращается только после того, как будут проверены все прототипы. Например, по запросу значения свойства creep у объекта yoshi поиск этого свойства инициируется сначала в объекте yoshi, как показано на рис. 7.3. А поскольку данное свойство отсутствует у объекта yoshi, то поиск продолжается в его прототипе hattori. Но и у прототипа hattori данное свойство отсутствует, поэтому поиск продолжается в его прототипе kuma, где оно обнаруживается в конечном итоге.

Итак, дав общее представление о механизме поиска конкретных свойств по цепочке прототипов, покажем, как пользоваться прототипами при создании новых объектов с помощью функций-конструкторов.

```
const yoshi = { skulk: true };
const hattori = { sneak: true };
const kuma = { creep: true };

Object.setPrototypeOf(yoshi, hattori);
Object.setPrototypeOf(hattori, kuma);
```

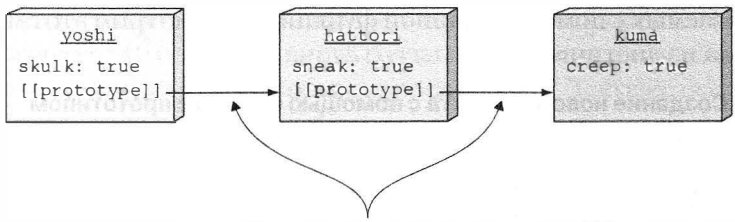


Рис. 7.3. Поиск конкретного свойства прекращается, когда все прототипы исчерпаны. По ссылке **yoshi.creep** инициируется поиск свойства **creep** сначала в объекте **yoshi**, затем в объекте **hattori** и, наконец, в объекте **kuma**

7.2. Создание объектов и прототипы

Создать новый объект проще всего с помощью оператора, подобного следующему:

```
const warrior = {};
```

В этом операторе создается новый пустой объект, который может быть затем наполнен свойствами с помощью операций присваивания, как показано ниже.

```
const warrior = {};
warrior.name = 'Saito';
warrior.occupation = 'marksman';
```

Но тем, у кого имеется опыт объектно-ориентированного программирования, может недоставать инкапсуляции и структурирования, сопутствующих понятию конструктора класса как функции, служащей для инициализации объекта в известное исходное состояние. Ведь если бы потребовалось создать несколько экземпляров объекта одного и того же типа, то присваивание многих свойств по отдельности оказалось бы не только трудоемкой, но и не лишенной ошибок операцией. Для этой цели требуются средства, позволяющие объединить свойства и методы для объектов класса в одном месте.

И такой механизм существует в JavaScript, хотя он и заметно отличается по своей форме от аналогичных механизмов в других языках программирования. Как и в других объектно-ориентированных языках программирования вроде Java и C++, в JavaScript реализована операция `new`, предназначенная для получения экземпляров новых объектов через конструкторы, но в то же время в JavaScript отсутствует определение класса как таковое. Вместо этого операция `new` применяется к функции-конструктору, как пояснялось в главе 3, инициируя тем самым создание в памяти нового объекта.

Но в предыдущих главах не пояснялось, что у каждой функции имеется свой объект-прототип , который автоматически задается в качестве прототипа объектов, создаваемых с помощью данной функции. Рассмотрим этот механизм на примере кода из листинга 7.2.

Листинг 7.2. Создание нового объекта с помощью метода с прототипом

```
function Ninja(){}
Ninja.prototype.swingSword = function(){
  return true;
};

const ninja1 = Ninja();
assert(ninja1 === undefined,
       "No instance of Ninja created.");

const ninja2 = new Ninja();
assert(ninja2 &&
       ninja2.swingSword &&
       ninja2.swingSword(),
       "Instance exists and method is callable." );
```

Определить функцию, которая ничего не делает и ничего не возвращает

У каждой функции имеется встроенный объект-прототип, который можно свободно видоизменять

Вызвать функцию как обычную функцию. Как показывает тест, при этом ничего особенного не происходит

Вызвать функцию как конструктор. Как показывает тест, при этом создается новый экземпляр объекта, получающий метод из прототипа функции

В данном примере кода сначала определяется функция `Ninja()`, которая, очевидно, ничего особенного не делает, но вызывается двумя способами. Как обычная функция:

```
const ninja1 = Ninja();
```

и как конструктор:

```
const ninja2 = new Ninja();
```

После создания этой функции ей сразу же назначается новый объект, присваиваемый ее объекту-прототипу, который может быть расширен подобно любому другому объекту. В данном случае он дополняется методом `swingSword()` следующим образом:

```
Ninja.prototype.swingSword = function(){
  return true;
};
```

Далее функция `Ninja()` подвергается проверке. Сначала эта функция вызывается как обычная функция, и результат сохраняется в переменной `ninja1`. Анализируя тело этой функции, можно заметить, что она вообще не возвращает значение, и поэтому значение переменной `ninja1`, скорее всего, должно быть неопределенным (`undefined`), что и подтверждает наш тест. Поэтому от функции `Ninja()`, какой бы простой она ни была, мало проку.

Затем функция `Ninja()` вызывается как *конструктор* через операцию `new`, и в этом случае все происходит совершенно иначе. Данная функция вызывается снова, но на этот раз в памяти создается новый объект, становящийся ее контекстом и доступный по ссылке `this`. Результат выполнения операции `new` возвращается в виде ссылки на этот новый объект. И далее проверяется, содержит ли переменная `ninja2` ссылку на вновь созданный объект, а сам объект — метод `swingSword()`, который можно вызвать впоследствии. Текущее состояние прикладного кода приведено на рис. 7.4.

Как видите, после создания функции, ей назначается новый объект, который присваивается ее свойству `prototype`. Первоначально у объекта-прототипа имеется единственное свойство `constructor`, содержащее обратную ссылку на данную функцию (мы еще вернемся к свойству `constructor` далее в этой главе).

Когда функция вызывается как конструктор, например, в операции `new Ninja()`, то в качестве прототипа вновь созданного объекта задается тот объект, на который делается ссылка в прототипе функции-конструктора. В данном примере свойство `Ninja.prototype` было расширено до метода `swingSword()`, а при создании объекта `ninja2` в его свойстве `prototype` указан прототип функции `Ninja`. Следовательно, при попытке получить доступ к свойству `swingSword` объекта `ninja2` поиск этого свойства выполняется в прототипе функции `Ninja`. Обратите внимание на то, что метод `swingSword()` доступен *всем* объектам, создаваемым конструктором объектов типа `Ninja`. И тем самым обеспечивается повторное использование кода!

```
function Ninja(){}
Ninja.prototype.swingSword = function(){
  return true;
};
…
const ninja2 = new Ninja();
```

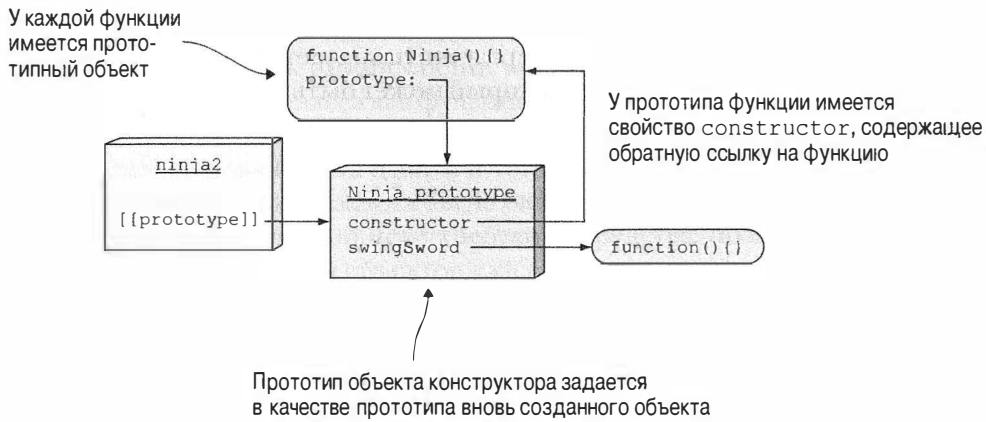


Рис. 7.4. Всякой функции при создании назначается новый объект-прототип. А когда она используется как конструктор, то в качестве прототипа вновь созданного объекта задается прототип этой функции

Метод `swingSword()` является свойством прототипа функции `Ninja`, а не экземпляров объекта `ninja`. Поэтому выясним, чем свойства экземпляров отличаются от свойств прототипов.

7.2.1. Свойства экземпляров

Когда функция вызывается как конструктор в операции `new`, в качестве ее контекста устанавливается новый экземпляр объекта. Это означает, что, помимо доступности свойств через прототип, имеется также возможность инициализировать значения новых свойств в функции-конструкторе через параметр `this`. Рассмотрим пример создания свойств такого экземпляра объекта в коде из листинга 7.3.

Листинг 7.3. Соблюдение приоритета операций инициализации

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
}
Ninja.prototype.swingSword = function(){
  return this.swung;
};
```

Создать переменную экземпляра, инициализируемую логическим значением `false`

Создать метод экземпляра, возвращающий обратное значение переменной экземпляра `swung`

Определить метод прототипа с тем же именем, что и у метода экземпляра. Какой же из них получит приоритет?

```
const ninja = new Ninja();
assert(ninja.swingSword(),
       "Called the instance method, not the prototype method.");
```

Создать экземпляр объекта типа `Ninja` и убедиться, что метод этого экземпляра переопределит одноименный метод прототипа

Приведенный выше код очень похож на код из предыдущего примера в том отношении, что метод `swingSword()` определяется путем его добавления к свойству `prototype` конструктора:

```
Ninja.prototype.swingSword = function(){
  return this.swung;
};
```

Но аналогично называемый метод вводится и в теле самой функции-конструктора (выделено ниже полужирным).

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
}
```

Оба метода определяются так, чтобы они возвращали противоположные значения, по которым можно было бы различить вызываемый метод.

Примечание

В реальном коде этого делать не рекомендуется. В данном примере это делается лишь для того, чтобы продемонстрировать приоритет свойств объекта.

Если выполнить тест, то он пройдет! Это наглядно показывает, что члены экземпляра объекта, создаваемые в конструкторе, перекрывают одноименные свойства, определяемые в прототипе (рис. 7.5).

В теле функции-конструктора ключевым словом `this` обозначается ссылка на вновь созданный объект, и поэтому свойства, введенные в конструкторе, создаются непосредственно для нового экземпляра объекта `ninja`. А когда требуется доступ к свойству `swingSword` объекта `ninja`, то обходить всю цепочку прототипов не нужно (см. рис. 7.4). Свойство, созданное в конструкторе, сразу же обнаруживается и возвращается, как показано на рис. 7.5.

Из этого вытекает интересный побочный эффект. В качестве примера на рис. 7.6 показано состояние прикладного кода, в котором созданы три экземпляра объекта `ninja`.

Как видите, каждому экземпляру объекта `ninja` назначается свой набор свойств, созданных в конструкторе, и все они имеют доступ к свойствам одного и того же прототипа. И это вполне разумно для свойств-значений (например, `swung`), специфичных для каждого экземпляра объекта. Но иногда это может создать затруднения при работе с методами.

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
}
Ninja.prototype.swingSword = function(){
  return this.swung;
};
const ninja = new Ninja();
```

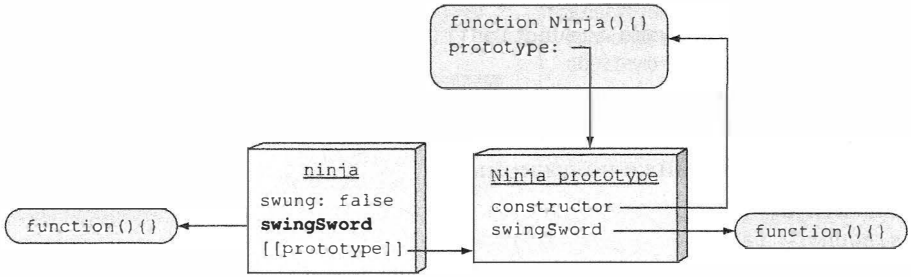


Рис. 7.5. Если свойство обнаруживается в самом экземпляре объекта, то обращаться к его прототипу не требуется!

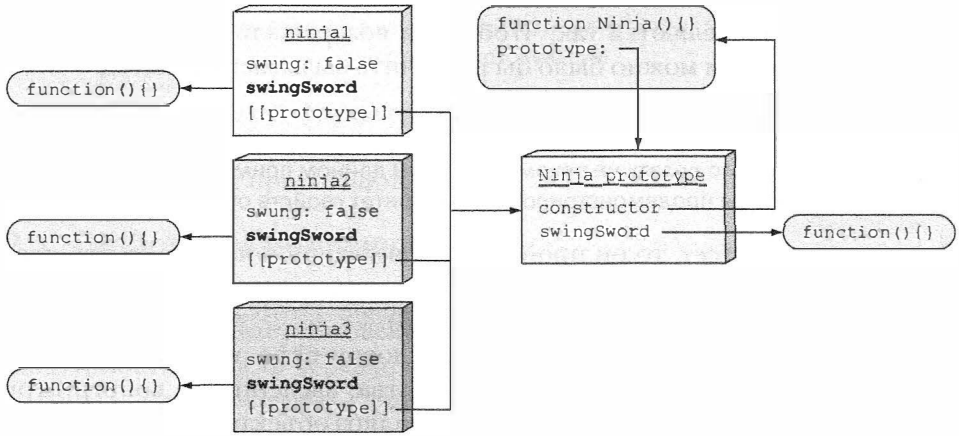


Рис. 7.6. Каждому экземпляру объекта назначается свой набор свойств, создаваемых в конструкторе, но все они имеют доступ к свойствам одного и того же прототипа

В данном примере мы столкнулись с тремя вариантами метода `swingSword()`, в которых реализована одна и та же логика. И это не вызывает особых проблем, если создается всего пара объектов, но требует особого внимания, если предполагается создать большое число объектов. Каждая копия метода ведет себя одинаково, и поэтому создавать многие его копии зачастую нецелесообразно из соображений рационального расходования оперативной памяти. Интерпретатор JavaScript может, конечно, выполнить определенную оптимизацию, но особенно полагаться на нее не стоит. С этой точки зрения целесоо-

бразно размещать методы объекта только в прототипе функции, поскольку в этом случае единственный метод станет общим для всех экземпляров объекта.

Примечание

Как упоминалось в главе 5, посвященной замыканиям, методы, определяемые в функциях-конструкторах, позволяют имитировать закрытые объектные переменные. Если требуется именно это, то методы следует указывать только в конструкторах.

7.2.2. Побочные эффекты динамического характера JavaScript

Как было показано ранее, JavaScript является динамическим языком, где свойства можно без особого труда добавлять, удалять и видоизменять по желанию. Это же относится и к прототипам как функций, так и объектов. Наглядный тому пример приведен в коде из листинга 7.4.

Листинг 7.4. Благодаря прототипам все может изменяться во время выполнения

```
function Ninja(){
  this.swung = true;
}
```

Определить конструктор, создающий объект типа `Ninja` с единственным свойством логического типа

```
const ninja1 = new Ninja();
```

Создать экземпляр объекта типа `Ninja`, вызвав функцию-конструктор в операции `new`

```
Ninja.prototype.swingSword = function(){
  return this.swung;
};
```

Добавить метод к прототипу после создания объекта

```
assert(ninja1.swingSword(),
       "Method exists, even out of order.");
```

Показать, что метод существует в объекте

```
Ninja.prototype = {
  pierce: function() {
    return true;
  }
}
```

Полностью заменить прототип `Ninja` новым объектом, содержащий метод `pierce()`

```
assert(ninja1.swingSword(),
       "Our ninja can still swing!");
```

Несмотря на то что прототип Ninja полностью заменен, наш ниндзя может по-прежнему размахивать мечом, поскольку в его объекте хранится ссылка на прежний прототип `Ninja`

```
const ninja2 = new Ninja();
assert(ninja2.pierce(),"Newly created ninjas can pierce");
assert(!ninja2.swingSword, "But they cannot swing!");
```

Вновь созданные объекты ниндзя ссылаются на новый прототип, и поэтому они могут колоть, но не размахивать мечом

И в этом примере кода определяется конструктор объектов типа `Ninja`, который далее используется для создания экземпляра объекта данного типа. Состояние прикладного кода в данный момент наглядно показано на рис. 7.7.

```
function Ninja(){
   this.swung = true;
}

const ninja1 = new Ninja();
```

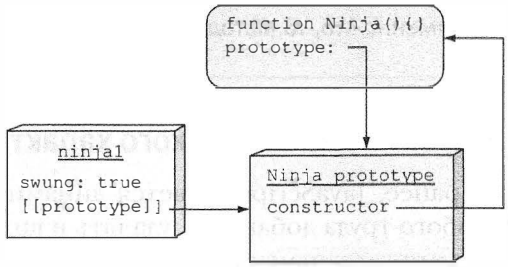


Рис. 7.7. После создания у объекта **ninja1** имеется свойство **swung**, а его прототипом является прототип **Ninja** с единственным свойством **constructor**

После создания экземпляра объекта в его прототип вводится метод `swingSword()`. Затем выполняется тест с целью продемонстрировать, что изменение, внесенное в прототип после создания объекта, возымело действие. Текущее состояние прикладного кода в данный момент наглядно показано на рис. 7.8.

```
function Ninja(){
  this.swung = true;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = function(){
  return this.swung;
};
```

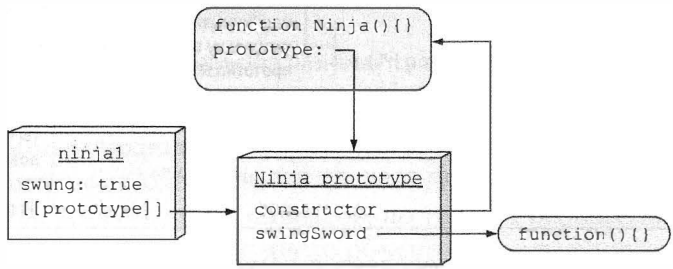


Рис. 7.8. Экземпляр **ninja1** ссылается на прототип **Ninja**, и поэтому доступны даже те изменения, которые были внесены после создания данного экземпляра

Далее прототип `Ninja` заменяется присваиванием ему совершенно нового объекта, имеющего метод `pierce()`. И это отражается на состоянии прикладного кода, как показано на рис. 7.9.

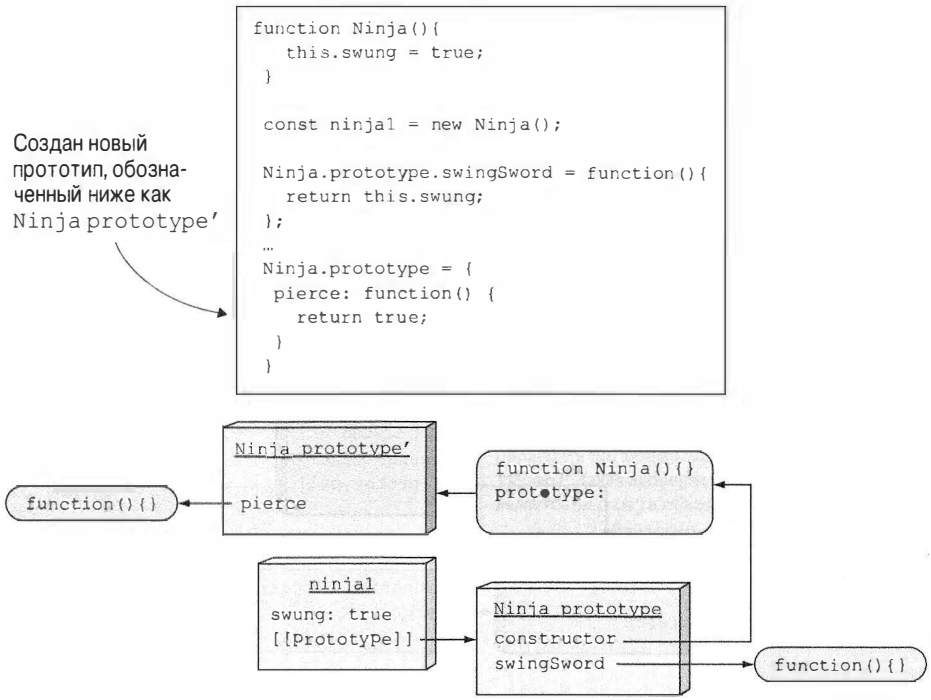


Рис. 7.9. Прототип функции можно заменить по желанию. Созданные ранее экземпляры будут ссылаться на прежний прототип!

Как видите, несмотря на то, что функция-конструктор объектов типа `Ninja` не ссылается на свой прежний прототип, он по-прежнему сохраняется активным в экземпляре `ninja1`, который все еще имеет доступ (по цепочке прототипов) к методу `swingSword()`. Но если создать новые объекты после столь неожиданной замены прототипов, то состояние прикладного кода будет таким, как на рис. 7.10.

Связь между объектом и прототипом функции-конструктора устанавливается в момент получения экземпляра данного объекта. Вновь созданные объекты будут ссылаться на новый прототип и получат доступ к методу `pierce()`, тогда как прежние объекты, созданные до замены прототипа, сохранят свой прототип, благополучно продолжая представлять ниндзя, размахивающих мечами.

Мы тщательно исследовали механизм действия прототипов и их связь с получением экземпляров объектов. Теперь можно сделать небольшой перерыв, чтобы продолжить исследование характера подобных объектов.

```
function Ninja(){
   this.swung = true;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = function(){
   return this.swung;
};
…
Ninja.prototype = {
 pierce: function() {
    return true;
 }
}
…
const ninja2 = new Ninja();
```

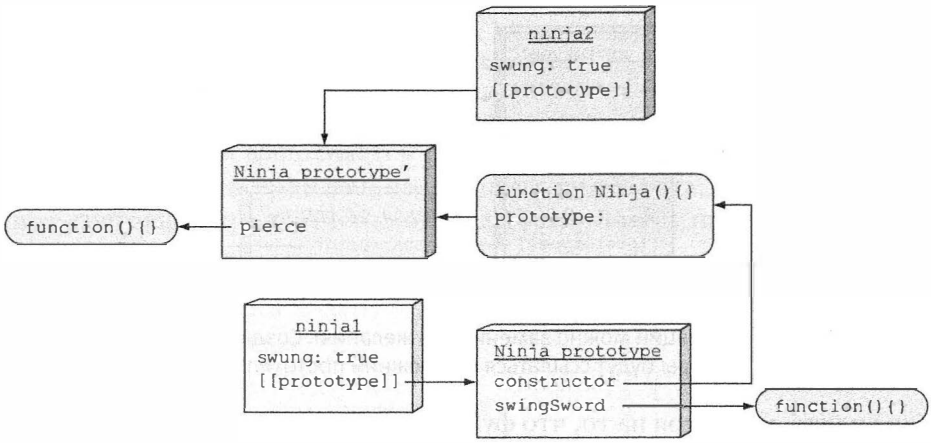


Рис. 7.10. Все вновь созданные экземпляры ссылаются на новый прототип

7.2.3. Типизация объектов через конструкторы

Полезно знать не только, каким образом интерпретатор JavaScript пользуется прототипом для поиска правильных ссылок на свойства, но и какая функция создала экземпляр объекта. Как было показано ранее, конструктор объекта доступен через свойство `constructor` прототипа функции-конструктора. В качестве примера на рис. 7.11 показано состояние прикладного кода при получении экземпляра с помощью конструктора объектов типа `Ninja`.

Используя свойство `constructor`, можно получить доступ к функции, с помощью которой был создан объект. Эта информация может служить в качестве формы контроля типов, как демонстрируется в примере кода из листинга 7.5.

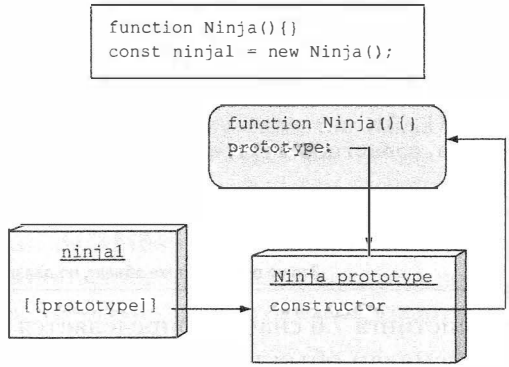


Рис. 7.11. У объекта-прототипа каждой функции имеется свойство **constructor**, ссылающееся на данную функцию

Листинг 7.5. Проверка типа экземпляра и его конструктора

```
function Ninja(){}
const ninja = new Ninja();

assert(typeof ninja === "object",
       "The type of the instance is object.");
assert(ninja instanceof Ninja,
       "instanceof identifies the constructor." );
assert(ninja.constructor === Ninja,
       "The ninja object was created by the Ninja function.");
```

Проверить тип экземпляра ninja в операции typeof. Такая проверка позволяет выяснить, является ли ninja объектом, но и только

Проверить тип экземпляра ninja в операции instanceof. Такая проверка дает больше информации, в частности, о том, что экземпляр был создан конструктором объектов типа Ninja

Проверить тип экземпляра ninja по ссылке на конструктор. Такая проверка дает ссылку на функцию-конструктор

В данном примере кода сначала определяется конструктор, и с его помощью создается экземпляр объекта. Затем с помощью операции `typeof` проверяется тип экземпляра этого объекта. Это не особенно помогает, поскольку все экземпляры будут распознаваться как объекты, и в результате всегда возвращается значение `"object"`. Намного более интересный результат дает операция `instanceof`, которая действительно помогает выяснить, был ли экземпляр объекта создан с помощью конкретной функции-конструктора. Подробнее об операции `instanceof` речь пойдет далее в этой главе.

Помимо этого, можно воспользоваться свойством `constructor`, которое, как известно, вводится во все экземпляры объектов, в качестве обратной ссылки на исходную функцию, создавшую экземпляр объекта. С помощью этого свойства можно проверить происхождение экземпляра объекта подобно тому, как это делается с помощью операции `instanceof`. А поскольку это всего лишь обратная ссылка на исходный конструктор, то по ней можно получить новый экземпляр объекта типа `Ninja`, как показано в примере кода из листинга 7.6.

Листинг 7.6. Получение нового экземпляра объекта по ссылке на конструктор

```
function Ninja(){}

const ninja = new Ninja();
const ninja2 = new ninja.constructor();

assert(ninja2 instanceof Ninja, "It's a Ninja!");
assert(ninja !== ninja2, "But not the same Ninja!");
```

Создать второй объект типа `Ninja` из первого

Убедиться в том, что новый экземпляр объекта относится к типу `Ninja`

Это не один и тот же объект, но два разных его экземпляра

В примере кода из листинга 7.6 сначала определяется конструктор, и с его помощью создается экземпляр объекта. Затем свойство `constructor` вновь созданного экземпляра объекта используется для создания второго экземпляра этого объекта. Как показывает проверка, второй экземпляр объекта типа `Ninja` отличается от первого.

В данном примере кода особое внимание обращает на себя тот факт, что экземпляр объекта можно получить, даже не обращаясь к исходной функции-конструктору. Ссылкой на этот конструктор можно пользоваться совершенно скрытно, даже если он уже не находится в текущей области видимости.

Примечание

Свойство **`constructor`** экземпляра объекта можно изменять, хотя очевидных или особых причин для этого не существует (разве что сделать это со злым умыслом). Ведь основное его назначение — уведомлять об источнике создания объекта. Если же свойство **`constructor`** перезаписывается, его исходное значение просто теряется.

Все это, конечно, очень хорошо и даже полезно знать, но мы затронули лишь малую часть тех огромных потенциальных возможностей, которые предоставляют нам прототипы. И самое интересное нас еще ждет впереди.

7.3. Достижение наследования

Наследование является формой повторного использования кода, в которой новые объекты получают доступ к свойствам существующих объектов. Благодаря наследованию исключается потребность повторять код и данные в кодовой базе. В языке JavaScript наследование действует несколько иначе, чем в других распространенных объектно-ориентированных языках программирования. Рассмотрим в качестве примера код из листинга 7.7, где предпринимается попытка достичь наследования.

Листинг 7.7. Попытка достичь наследования с помощью прототипов

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
```

Определить объект типа `Person`, представляющий танцующего человека, через его конструктор и прототип

Определить объект типа `Ninja`, представляющий ниндзя

```
Ninja.prototype = { dance: Person.prototype.dance };

const ninja = new Ninja();
assert(ninja instanceof Ninja,
       "ninja receives functionality from the Ninja prototype" );
assert(ninja instanceof Person, "... and the Person prototype" );
assert(ninja instanceof Object, "... and the Object prototype" );
```

Попытаться сделать ниндзя танцующим человеком, скопировав метод `dance()` из прототипа `Person`

Прототип функции является обычным объектом, поэтому существует несколько способов копирования его функциональных возможностей, в том числе свойств и методов, чтобы осуществить наследование. В приведенном выше примере кода сначала определяется объект типа `Person`, а затем объект типа `Ninja`. А поскольку объект типа `Ninja` явно определяет человека (в данном случае ниндзя), то можно сделать попытку добиться того, чтобы он унаследовал свойства объекта типа `Person`, определяющего человека вообще. И такая попытка предпринимается путем копирования свойства `dance` из метода прототипа `Person` в одноименное свойство прототипа `Ninja`.

Как показывает тестирование рассматриваемого здесь кода (рис. 7.12), объект типа `Ninja` нельзя сделать объектом типа `Person`. И хотя ниндзя можно научить танцевать, как и всякого человека, объект типа `Ninja` все равно нельзя сделать объектом типа `Person`, скопировав соответствующее свойство. Ведь это не наследование, а только копирование.

Такой подход не вполне пригоден, поскольку он предполагает лишь ручное копирование каждого свойства в отдельности из прототипа одного объекта (`Person`) в прототип другого объекта (`Ninja`). Но ведь это не имеет никакого отношения к наследованию. Поэтому продолжим исследование данного вопроса.

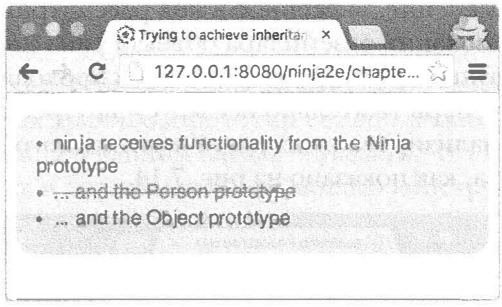


Рис. 7.12. Объект типа **`Ninja`** на самом деле не является объектом типа **`Person`**. Поэтому особой радости танцы не приносят!

На самом деле нам требуется образовать *цепочку прототипов*, чтобы определить ниндзя (объект тип `Ninja`) как человека (объект типа `Person`), человека (объект типа `Person`) — как млекопитающее (объект типа `Mammal`), а млекопитающее (объект типа `Mammal`) — как животное (объект типа `Animal`), и так далее до самого объекта типа `Object`. Такую цепочку прототипов лучше всего

создать, используя экземпляр одного объекта в качестве прототипа другого объекта:

```
SubClass.prototype = new SuperClass();
```

Например:

```
Ninja.prototype = new Person();
```

В этом случае цепочка прототипов сохраняется, поскольку прототип экземпляра класса `SubClass` будет экземпляром класса `SuperClass`, у которого имеется прототип со всеми свойствами объекта типа `SuperClass`, а у того, в свою очередь, — прототип, указывающий на экземпляр *его* суперкласса, и т.д. Попробуем применить такой способ построения цепочки прототипов, внеся незначительные изменения в пример кода из листинга 7.7, как выделено полужирным в листинге 7.8.

Листинг 7.8. Достижение наследования с помощью прототипов

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

const ninja = new Ninja();
assert(ninja instanceof Ninja,
       "ninja receives functionality from the Ninja prototype");
assert(ninja instanceof Person, "... and the Person prototype");
assert(ninja instanceof Object, "... and the Object prototype");
assert(typeof ninja.dance === "function", "... and can dance!")
```

Сделать объект типа `Ninja` объектом типа `Person`, превратив прототип `Ninja` в экземпляр объекта типа `Person`

Единственное изменение, внесенное в код рассматриваемого здесь примера, состоит в использовании экземпляра объекта типа `Person` в качестве прототипа для объекта типа `Ninja`. В итоге все тесты проходят успешно, как показано на рис. 7.13. А теперь тщательно исследуем внутренний механизм такого наследования, проанализировав состояние прикладного кода после создания нового объекта `ninja`, как показано на рис. 7.14.

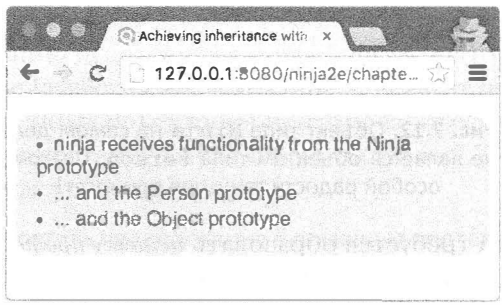


Рис. 7.13. Теперь объект типа **`Ninja`** стал объектом типа **`Person`**. На радостях можно и сплясать победный танец!

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();
const ninja = new Ninja();
```

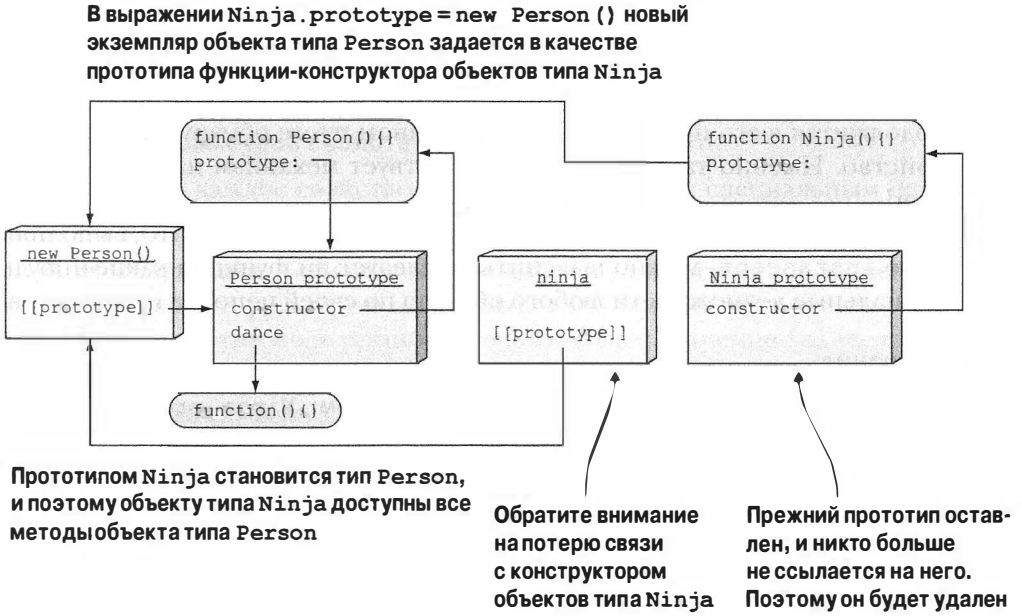


Рис. 7.14. Мы достигли наследования, задав в прототипе функции-конструктора объектов типа **Ninja** новый экземпляр объекта типа **Person**

Как показано на рис. 7.14, когда определяется функция `Person()`, создается также прототип `Person`, ссылающийся на функцию `Person()` через свое свойство `constructor`. Как правило, прототип `Person` можно расширить дополнительными свойствами, а в данном случае — указать, что каждый объект, представляющий человека и создаваемый с помощью конструктора объектов типа `Person`, получает доступ к методу `dance()`:

```
function Person(){}
Person.prototype.dance = function(){};
```

В рассматриваемом здесь примере кода определяется также функция `Ninja()`, которая получает свой объект-прототип со свойством `constructor`, ссылающимся на эту функцию следующим образом:

```
function Ninja(){}
```

Для достижения наследования прототип функции `Ninja()` заменяется далее новым экземпляром объекта `Person`. Если теперь создать новый объект типа `Ninja`, во внутреннем свойстве `prototype` вновь созданного объекта

ninja будет задан объект, на который указывает свойство prototype текущего прототипа Ninja, т.е. на созданный ранее экземпляр типа Person:

```
function Ninja(){}
Ninja.prototype = new Person();
var ninja = new Ninja();
```

При попытке получить доступ к методу dance() через объект ninja интерпретатор JavaScript проверит сначала сам объект ninja. А поскольку у него отсутствует свойство dance, то поиск продолжится в его прототипе (объекте person). У объекта person также отсутствует свойство dance, и поэтому поиск продолжится в его прототипе, где в конечном итоге обнаруживается данное свойство. Именно таким образом и действует механизм наследования в JavaScript!

Самое важное следствие из этого примера заключается в том, что, выполняя операцию instanceof, можно выяснить, наследует ли функция какие-нибудь функциональные возможности любого объекта по своей цепочке прототипов.

Примечание

Не рекомендуется пользоваться следующим способом: **Ninja.prototype = Person.prototype;**, т.е. указывая прототип **Person** непосредственно как прототип **Ninja**. Ведь в этом случае любые изменения в прототипе **Ninja** обусловят изменения в прототипе **Person**, поскольку они представляют один и тот же объект. А это может привести к нежелательным побочным эффектам.

Еще один побочный эффект от такого наследования прототипов состоит в том, что все наследуемые прототипы функций можно будет и далее обновлять. Объекты, унаследованные от прототипа, всегда имеют доступ к свойствам текущего прототипа.

7.3.1. Трудности переопределения свойства constructor

Если внимательнее рассмотреть рис. 7.14, то можно заметить, что, задавая новый объект типа Person в качестве прототипа функции-конструктора объектов типа Ninja, мы теряем связь с конструктором объектов типа Ninja, которая раньше поддерживалась в первоначальном прототипе Ninja. Это создает определенные трудности, поскольку свойство constructor может быть использовано для определения функции, с помощью которой был создан объект. Если кто-нибудь попытается воспользоваться рассматриваемым здесь кодом, то он может вполне обоснованно допустить, что следующий тест пройдет:

```
assert(ninja.constructor === Ninja,
       "The ninja object was created by the Ninja constructor");
```

Однако в текущем состоянии прикладного кода этот тест не пройдет. Как показано на рис. 7.14, всякая попытка найти свойство constructor в объекте ninja завершится неудачно. Следовательно, поиск продолжается в его прототипе, у которого также отсутствует свойство constructor, и поэтому он пере-

ходит далее к объекту-прототипу типа Person, где и обнаруживается свойство constructor, ссылающееся на функцию-конструктор объектов типа Person. По существу, если попытаться выяснить, какая функция создала объект ninja, то в итоге окажется, что он был создан функцией-конструктором объектов типа Person, что неверно. И это может стать источником некоторых серьезных программных ошибок.

Исправить это положение мы должны сами! Но прежде чем сделать это, нужно выяснить, как в JavaScript выполняется настройка параметров свойств объекта.

Настройка параметров свойств объектов

В JavaScript каждое свойство объекта описывается специальным *дескриптором свойства*, через который могут быть изменены следующие поля.

- **configurable**. Если в этом поле установлено логическое значение true, то дескриптор свойства может быть изменен, а само свойство — удалено. Если же в этом поле установлено логическое значение false, то ни одно из этих действий выполнить нельзя.
- **enumerable**. Если в этом поле установлено логическое значение true, то свойство проявляется при переборе свойств объекта в цикле for-in (более подробно цикл for-in рассматривается далее в этой главе).
- **value**. В этом поле определяется значение свойства. По умолчанию в нем устанавливается неопределенное значение (undefined).
- **writable**. Если в этом поле установлено логическое значение true, то значение свойства может быть изменено с помощью операции присваивания.
- **get**. В этом поле определяется *метод получения*, который будет вызываться при доступе к данному свойству. Данное поле нельзя определить вместе с полями value и writable.
- **set**. В этом поле определяется *метод установки*, который будет вызываться всякий раз, когда свойству требуется присвоить конкретное значение. Данное поле нельзя определить вместе с полями value и writable.

Предположим, мы создали свойство с помощью обычной операции присваивания, как показано в следующем примере кода:

```
ninja.name = "Yoshi";
```

Данное свойство будет настраиваемым (configurable), перечисляемым (enumerable) и доступным по записи (writable). Его поло value будет присвоено значение "Yoshi", а методы get() и set() окажутся неопределенными (undefined).

Если же требуется выполнить тонкую настройку параметров свойства, то для этой цели можно воспользоваться встроенным методом Object.defineProperty(). Ему передается объект, в котором будет определено свой-

ство, имя свойства, а также объект-дескриптор свойства. В качестве примера рассмотрим код из листинга 7.9.

Листинг 7.9. Настройка параметров свойств объекта

```
var ninja = {};
ninja.name = "Yoshi";
ninja.weapon = "kusarigama";

Object.defineProperty(ninja, "sneaky", {
  configurable: false,
  enumerable: false,
  value: true,
  writable: true
});

assert("sneaky" in ninja, "We can access the new property");

for(let prop in ninja){
  assert(prop !== undefined, "An enumerated property: " + prop);
}
```

Создать пустой объект; воспользоваться операциями присваивания для определения двух свойств

Встроенный метод `Object.defineProperty()` служит для тонкой настройки параметров свойства объекта

Перебрать перечисляемые свойства объекта `ninja` в цикле `for-in`

Сначала в данном примере кода создается пустой объект, в который добавляется два свойства, `name` и `weapon`, старым, проверенным способом, т.е. с помощью операции присваивания. Затем вызывается встроенный метод `Object.defineProperty()`, который определяет свойство `sneaky` как ненастраиваемое и неперечислимое в полях `configurable` и `enumerable`, а в его поле `value` устанавливается логическое значение `true`. Значение в этом поле можно изменить, поскольку данное свойство определяется как доступное по записи в поле `writable`.

И, наконец, в данном примере кода проверяется возможность доступа к вновь созданному свойству `sneaky`, после чего все перечислимые свойства объекта выявляются в цикле `for-in`. Результат выполнения кода из данного примера приведен на рис. 7.15.

Установив логическое значение `false` в поле `enumerable`, можно сделать свойство недоступным для цикла `for-in`. Чтобы понять причину, придется вернуться к первоначальному затруднению, связанному с переопределением свойств.

Окончательное устранение трудностей при переопределении свойств

При попытке расширить объект типа `Person` с помощью объекта типа `Ninja` (или сделать класс `Ninja` подклассом, производным от класса `Person`) возникает следующее затруднение: если задать новый объект типа `Person` в качестве прототипа функции-конструктора объектов типа `Ninja`, то утратится связь с первоначальным прототипом `Ninja`, хранящим свойство `constructor`. Но это свойство нельзя терять, поскольку с его помощью удобно определяется

функция, применяемая для создания экземпляров объекта типа `Ninja`, на что вполне обоснованно рассчитывают другие разработчики, использующие вашу кодовую базу.

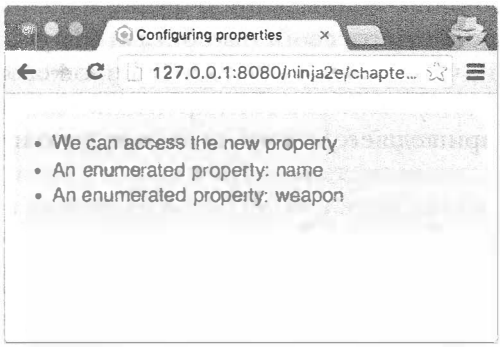


Рис. 7.15. В цикле **`for-in`** просматриваются свойства **`name`** и **`weapon`**, но не специально введенное свойство **`sneaky`**, хотя оно и доступно обычным способом

Подобное затруднение можно разрешить, воспользовавшись тем, что нам уже известно о свойствах. С этой целью определим новое свойство `constructor` для нового объекта `Ninja.prototype`, вызвав метод `Object.defineProperty()`, как показано в примере кода из листинга 7.10.

Листинг 7.10. Устранение трудностей при переопределении свойства `constructor`

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", {
  enumerable: false,
  value: Ninja,
  writable: true
});

var ninja = new Ninja();

assert(ninja.constructor === Ninja,
       "Connection from ninja instances to Ninja constructor
        reestablished!");
for(let prop in Ninja.prototype){
  assert(prop === "dance",
         "The only enumerable property is dance!");
}
```

Определить новое неперечислимое свойство `constructor`, снова указывающее на прототип `Ninja`

Восстановить связь

В `Ninja.prototype` не было добавлено никаких перечислимых свойств

Если теперь выполнить рассматриваемый здесь код, то все в нем окажется замечательно. В частности, нам удалось восстановить связь экземпляров объекта `ninja` с функцией-конструктором объектов типа `Ninja`, и таким образом мы можем узнать, что они созданы данной функцией-конструктором. И если кто-то попытается просмотреть свойства объекта `Ninja.prototype` в цикле, то мы гарантировали, что специально определенное свойство `constructor` окажется при этом недоступным. И в этом проявляются отличительные черты настоящего ниндзя, пришедшего, сделавшего свое дело и ушедшего никем незамеченным!

7.3.2. Операция `instanceof`

В большинстве языков программирования самый простой способ проверить, относится ли объект к определенной иерархии классов, состоит в том, чтобы воспользоваться операцией `instanceof`. Например, в языке Java с помощью операции `instanceof` проверяется, относится ли объект, находящийся в левой части данной операции, к тому же самому типу класса или подкласса, который указан в его правой ее части.

Несмотря на определенные параллели, которые можно было бы провести с принципом действия операции `instanceof` в JavaScript, сделать это не так-то просто. Ведь в языке JavaScript операция `instanceof` действует по цепочке прототипов. Допустим, имеется следующее выражение:

```
ninja instanceof Ninja
```

В операции `instanceof` проверяется, находится ли *текущий* прототип функции-конструктора объектов типа `Ninja` в цепочке прототипов экземпляра `ninja`. Обратимся за конкретным примером снова к людям вообще и ниндзя в частности, как демонстрируется в коде из листинга 7.11.

Листинг 7.11. Исследование операции `instanceof`

```
function Person(){}
function Ninja(){}

Ninja.prototype = new Person();

const ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");
assert(ninja instanceof Person, "A ninja is also a Person. ");
```

Экземпляр `ninja` относится как к типу `Ninja`, так и к типу `Person`

Как и предполагалось, экземпляр `ninja` одновременно относится и к типу `Ninja` и к `Person`. На рис. 7.16 показано состояние прикладного кода, раскрывающее внутренний механизм действия цепочки прототипов.

Цепочка прототипов экземпляра `ninja` состоит из объекта, который создается в результате выполнения операции `new Person()` и через который достигается наследование, а также прототипа `Person`. При вычислении значения вы-

ражения `ninja instanceof Ninja`, интерпретатор JavaScript берет прототип функции `Ninja` (т.е. объект, полученный в результате выполнения операции `new Person()`) и проверяет, находится ли этот объект в цепочке прототипов экземпляра `ninja`. А поскольку объект, создаваемый в операции `new Person()`, является непосредственным прототипом экземпляра `ninja`, то результат данной проверки оказывается истинным.

Во втором случае, когда вычисляется значение выражения `ninja instanceof Person`, интерпретатор JavaScript берет прототип функции-конструктора объектов типа `Person` и проверяет, находится ли этот прототип в цепочке прототипов экземпляра `ninja`. И в этом случае проверка проходит успешно, поскольку мы имеем дело с прототипом объекта, который создан в результате выполнения операции `new Person()` и является прототипом экземпляра `ninja`, как было выяснено выше.

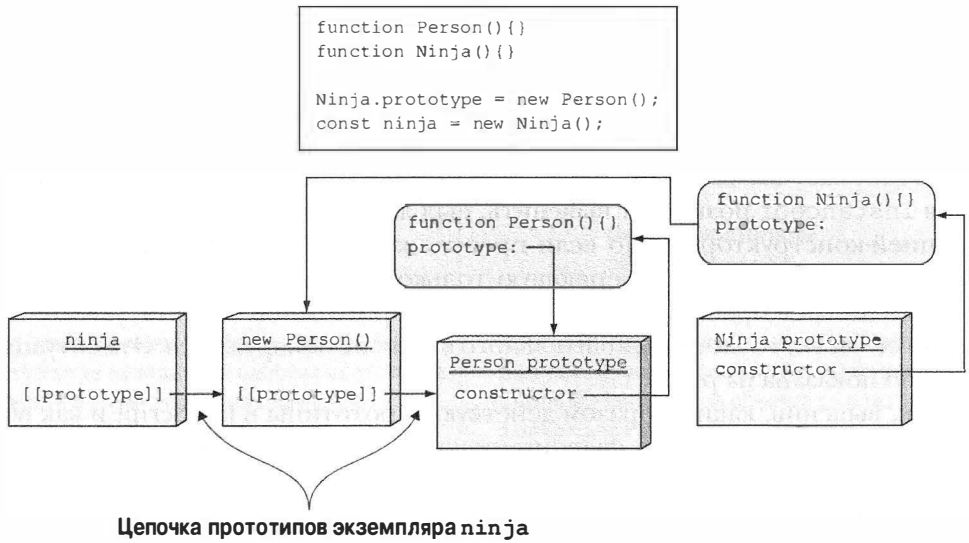


Рис. 7.16. Цепочка прототипов экземпляра **`ninja`** состоит из объекта, создаваемого в операции **`new Person()`**, а также прототипа **`Person`**

И это все, что следовало бы знать об операции `instanceof`. И хотя эта операция чаще всего применяется с целью выяснить, был ли экземпляр создан конкретной функцией-конструктором, на самом деле она действует несколько иначе. Вместо этого в операции `instanceof` проверяется, находится ли прототип функции, указываемой в правой части данной операции, в цепочке прототипов объекта, указываемого в левой ее части. Следовательно, применение операции `instanceof` требует дополнительного разъяснения.

Разъяснение операции `instanceof`

Как не раз упоминалось ранее в этой главе, JavaScript является динамическим языком, допускающим немало видоизменений во время выполнения про-

граммы. Ничто, например, не мешает изменить по ходу дела прототип конструктора, как демонстрируется в примере кода из листинга 7.12.

Листинг 7.12. Наблюдение за изменениями прототипов конструктора

```
function Ninja(){}

const ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");

Ninja.prototype = {};

assert(!(ninja instanceof Ninja),
       "The ninja is now not a Ninja!?");
```

Изменить прототип функции-конструктора объектов типа Ninja

Несмотря на то что экземпляр ninja был создан конструктором объектов типа Ninja, из операции instanceof теперь следует, что ninja вообще не является экземпляром типа Ninja!

В данном примере кода снова повторяются все основные шаги по созданию экземпляра `ninja`, и поэтому первый тест успешно проходит. Но если изменить прототип функции-конструктора объектов типа `Ninja` *после* создания экземпляра `ninja` и снова проверить, относится ли экземпляр `ninja` к типу `Ninja` (в выражении `instanceof Ninja`), то окажется, что положение изменилось. Это может вызвать удивление, если прийти к неверному выводу, что операция `instanceof` позволяет выяснить, был ли экземпляр создан конкретной функцией-конструктором. Но если проанализировать настоящую семантику операции `instanceof`, проверяющую только, находится ли прототип функции, указанный в правой ее части, в цепочке прототипов объекта, указанного в левой ее части, то ничего удивительного в этом не обнаружится. Эта ситуация наглядно показана на рис. 7.17.

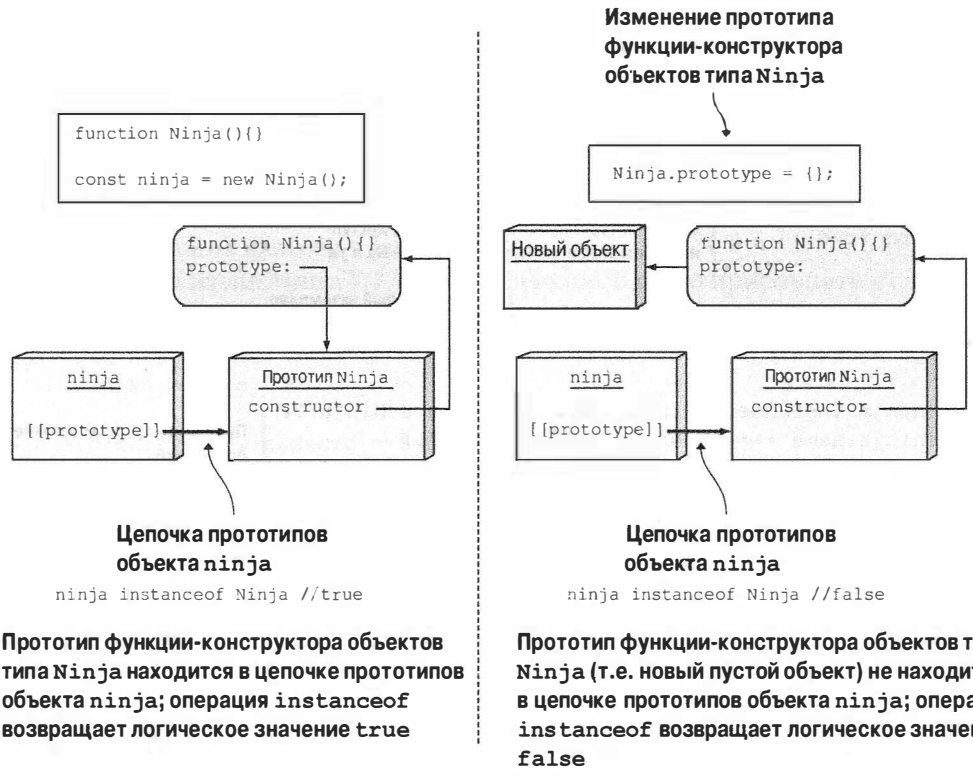
Итак, выяснив, каким образом действуют прототипы в JavaScript и как воспользоваться ими вместе с функциями-конструкторами для реализации наследования, перейдем к рассмотрению классов — очередному нововведению в стандарте ES6 языка JavaScript.

7.4. Применение "классов" в стандарте ES6 языка JavaScript

То, что JavaScript позволяет организовать наследование в определенной форме с помощью прототипов, само по себе замечательно. Но общим желанием многих разработчиков, особенно с опытом программирования в классическом объектно-ориентированном стиле, является упрощение или абстракция системы наследования в JavaScript до более знакомого им уровня.

Это неизбежно приводит к такому понятию, как классы, хотя в JavaScript классическая поддержка наследования в естественном виде не реализована. В ответ на эти потребности появился ряд библиотек JavaScript, где наследование имитируется в классическом виде. Но поскольку в каждой библиотеке классы реализуются по-своему, комитет по развитию стандарта ECMAScript

стандартизировал синтаксис для имитации наследования на основе классов. Обратите внимание на слово "имитация". Несмотря на то что, программируя на JavaScript, теперь можно пользоваться ключевым словом `class`, базовая реализация наследования по-прежнему основывается на прототипах!



Прототип функции-конструктора объектов типа `Ninja` находится в цепочке прототипов объекта `ninja`; операция `instanceof` возвращает логическое значение `true`

Прототип функции-конструктора объектов типа `Ninja` (т.е. новый пустой объект) не находится в цепочке прототипов объекта `ninja`; операция `instanceof` возвращает логическое значение `false`

Рис. 7.17. В операции `instanceof` проверяется, находится ли прототип функции, указанный в правой ее части, в цепочке прототипов объекта, указанного в левой ее части. Будьте внимательны, поскольку прототип функции может измениться в любой момент!

Примечание

Ключевое слово `class` было введено в стандарт ES6 языка JavaScript, но оно поддерживается еще не во всех браузерах (подробнее о текущей поддержке ключевого слова `class` в браузерах см. по адресу `http://kangax.github.io/compat-table/es6/#test-class`).

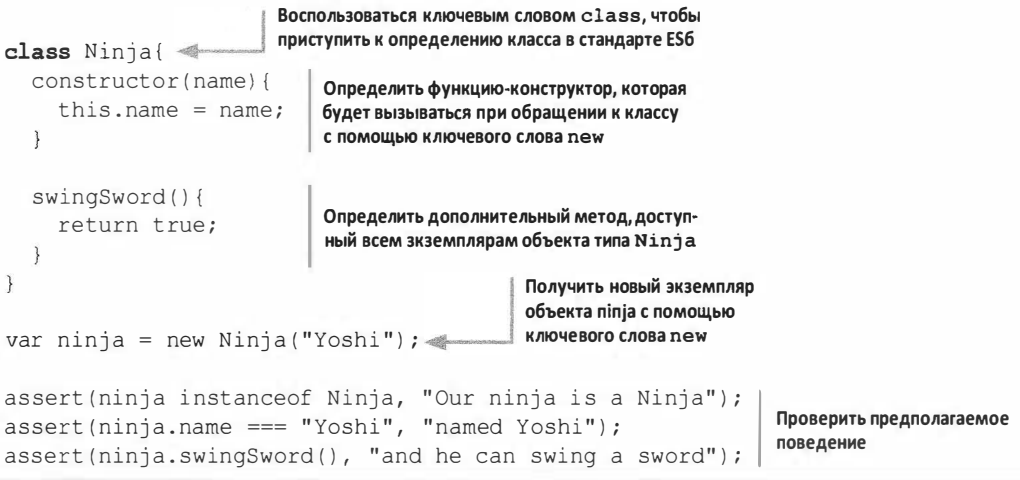
Итак, приступим к рассмотрению нового синтаксиса наследования на основе классов.

7.4.1. Применение ключевого слова `class`

Как упоминалось выше, в стандарт ES6 языка JavaScript было введено ключевое слово `class`, предоставляющее намного более изящный способ создания

объектов и реализации наследования, чем его реализация вручную с помощью прототипов. Пользоваться ключевым словом `class` нетрудно, как демонстрируется в примере кода из листинга 7.13.

Листинг 7.13. Создание класса в стандарте ES6




```
class Ninja{
  constructor(name){
    this.name = name;
  }

  swingSword(){
    return true;
  }
}

var ninja = new Ninja("Yoshi");

assert(ninja instanceof Ninja, "Our ninja is a Ninja");
assert(ninja.name === "Yoshi", "named Yoshi");
assert(ninja.swingSword(), "and he can swing a sword");
```

В примере кода из листинга 7.13 класс `Ninja` создается с помощью ключевого слова `class`. При создании классов в стандарте ES6 можно явным образом определить функцию `constructor()`, которая будет вызываться при получении экземпляра объекта (в данном случае типа `Ninja`). В теле конструктора можно получить доступ к вновь созданному экземпляру с помощью ключевого слова `this` и без особого труда ввести новые свойства вроде `name`, а в теле самого класса — определить методы, которые станут доступными всем экземплярам этого класса. В данном случае в теле класса `Ninja` определен метод `swingSword()`, возвращающий логическое значение `true`:

```
class Ninja{
  constructor(name){
    this.name = name;
  }
  swingSword(){
    return true;
  }
}
```

Далее в рассматриваемом здесь примере кода для создания экземпляра класса `Ninja` вызывается его конструктор с помощью ключевого слова `new` подобно тому, как это делалось в предыдущих примерах кода с помощью простой функции-конструктора и показано ниже.

```
var ninja = new Ninja("Yoshi");
```

И, наконец, в данном примере кода проверяется предполагаемое поведение экземпляра ninja. В частности, его принадлежность к типу Ninja (в операции instanceof Ninja), наличие свойства name и доступа к методу swingSword():

```
assert(ninja instanceof Ninja, "Our ninja is a Ninja");
assert(ninja.name === "Yoshi", "named Yoshi");
assert(ninja.swingSword(), "and he can swing a sword")
```

Классы как синтаксическое удобство

Как упоминалось ранее, несмотря на введение ключевого слова class в стандарт ES6, интерпретатор JavaScript по-прежнему работает на основе прототипов. А классы служат лишь синтаксическим удобством, упрощающим имитацию настоящих классов в JavaScript.

Код класса из листинга 7.13 можно функционально приравнять к следующему коду в стандарте ES5:

```
function Ninja(name) {
  this.name = name;
}
Ninja.prototype.swingSword = function() {
  return true;
};
```

Как видите, в новых классах, появившихся в стандарте ES6, нет ничего особенного. Их код выглядит более изящно, хотя он и основывается на тех же принципах и понятиях, что и код в стандарте ES5.

Статические методы

В предыдущих примерах было показано, каким образом определяются методы объектов (прототипные методы), доступные всем экземплярам объектов. Помимо таких методов, в классических объектно-ориентированных языках программирования вроде Java применяются статические методы, определяемые на уровне классов. В качестве примера рассмотрим код из листинга 7.14.

Листинг 7.14. Статические методы в стандарте ES6

```
class Ninja{
  constructor(name, level){
    this.name = name;
    this.level = level;
  }

  swingSword() {
   return true;
  }

  static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
  }
}
```

Воспользоваться ключевым словом static, чтобы определить статический метод

```
var ninja1 = new Ninja("Yoshi", 4);
var ninja2 = new Ninja("Hattori", 3);

assert(!("compare" in ninja1) && !("compare" in ninja2),
       "A ninja instance doesn't know how to compare");

assert(Ninja.compare(ninja1, ninja2) > 0,
       "The Ninja class can do the comparison!");

assert(!("swingSword" in Ninja),
       "The Ninja class cannot swing a sword");
```

Экземпляры `ninja` не имеют доступа к методу `compare()`

У класса `Ninja` имеется доступ к методу `compare()`

И в данном примере кода создается класс `Ninja` с методом `swingSword()`, доступным всем экземплярам `ninja`. В нем также определяется статический метод `compare()`, о чем свидетельствует ключевое слово `static`, указанное перед его именем:

```
static compare(ninja1, ninja2){
  return ninja1.level - ninja2.level;
}
```

Статический метод `compare()`, сравнивающий уровни мастерства двух ниндзя, определяется на уровне класса, а не экземпляра! А далее проверяется, что метод, по существу, недоступен из экземпляра `ninja`, но доступен из класса `Ninja`:

```
assert(!("compare" in ninja1) && !("compare" in ninja2),
       "The ninja instance doesn't know how to compare");
assert(Ninja.compare(ninja1, ninja2) > 0,
       "The Ninja class can do the comparison!");
```

Статические методы могут быть реализованы и в коде, где стандарт ES6 не используется. Но для этого придется вспомнить, что классы реализуются через функции. А поскольку статические методы действуют на уровне класса, то их можно реализовать, выгодно воспользовавшись функциями в качестве объектов высшего порядка и введя свойство метода в функцию-конструктор, как показано в следующем примере кода:

```
function Ninja(){}
Ninja.compare = function(ninja1, ninja2){...}
```

Расширить функцию-конструктор методом для имитации статических методов в коде, где не спользуется стандарт ES6

А теперь перейдем к наследованию.

7.4.2. Реализация наследования

Откровенно говоря, осуществить наследование в коде до ES6 не так-то просто. Обратимся за конкретным примером снова к людям, представленным объектами типа `Person`, и ниндзя, представленным объектами типа `Ninja`, как показано в следующем фрагменте кода:

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", {
  enumerable: false,
  value: Ninja,
  writable: true
});
```

В данном коде обращает на себя внимание следующее: методы, доступные экземплярам, должны быть непосредственно введены в прототип функции-конструктора, как это сделано с методом `dance()` и конструктором объектов типа `Person`. Если же требуется реализовать наследование, то придется задать прототип производного класса для наследования базового класса. В данном случае новый экземпляр типа `Person` присваивается объекту типа `Ninja.prototype`. К сожалению, в результате этой операции теряется свойство `constructor` прототипа, и поэтому его приходится восстанавливать с помощью метода `Object.defineProperty()`. Как видите, пытаясь достичь такого относительно простого и распространенного языкового средства, как наследование, необходимо принимать во внимание многие факторы. Правда, начиная со стандарта ES6, дело значительно упростилось.

В примере кода из листинга 7.15 показано, насколько просто можно теперь реализовать наследование в JavaScript.

Листинг 7.15. Наследование в стандарте ES6

```
class Person {
  constructor(name){
    this.name = name;
  }

  dance(){
    return true;
  }
}

class Ninja extends Person {
  constructor(name, weapon){
    super(name);
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}

var person = new Person("Bob");
```

Воспользоваться ключевым словом `extends`, чтобы наследовать от другого класса

Воспользоваться ключевым словом `super`, чтобы вызвать конструктор базового класса

```
assert(person instanceof Person, "A person's a person");
assert(person.dance(), "A person can dance.");
assert(person.name === "Bob", "We can call it by name.");
assert(!(person instanceof Ninja), "But it's not a Ninja");
assert(!("wieldWeapon" in person), "And it cannot wield a weapon");

var ninja = new Ninja("Yoshi", "Wakizashi");

assert(ninja instanceof Ninja, "A ninja's a ninja");
assert(ninja.wieldWeapon(), "That can wield a weapon");
assert(ninja instanceof Person, "But it's also a person");
assert(ninja.name === "Yoshi" , "That has a name");
assert(ninja.dance(), "And enjoys dancing");
```

В примере кода из листинга 7.15 демонстрируется, каким образом наследование достигается в стандарте ES6. В частности, для наследования одного класса от другого применяется ключевое слово `extends`, как показано в следующей строке кода:

```
class Ninja extends Person
```

В данном примере кода создается класс `Person` с конструктором, в котором свойству `name` каждого экземпляра класса `Person` присваивается значение `name`, как показано ниже. В этом классе определяется также метод `dance()`, который становится доступным всем экземплярам класса `Person`.

```
class Person {
  constructor(name){
    this.name = name;
  }
  dance(){
    return true;
  }
}
```

А далее в рассматриваемом здесь примере кода создается класс `Ninja`, расширяющий класс `Person`, как выделено ниже полужирным шрифтом. У него имеется дополнительное свойство `weapon` и метод `wieldWeapon()`.

```
class Ninja extends Person {
  constructor(name, weapon){
    super(name);
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}
```

В конструкторе производного класса `Ninja` с помощью ключевого слова `super` вызывается конструктор базового класса `Person`. Такая форма наследо-

вания должна быть вам знакома, если у вас имеется опыт программирования на любом из языков, основанных на классах.

После этого в данном примере кода создается экземпляр `person` и проверяется следующее: относится ли он к классу `Person`, имеет ли имя (`name`) человек, которого он представляет, и может ли этот человек танцевать (методом `dance()`). На всякий случай проверяется также, что человек, *не* относящийся к типу ниндзя (`Ninja`), не может владеть оружием:

```
var person = new Person("Bob");

assert(person instanceof Person, "A person's a person");
assert(person.dance(), "A person can dance.");
assert(person.name === "Bob", "We can call it by name.");
assert(!(person instanceof Ninja), "But it's not a Ninja");
assert(!("wieldWeapon" in person), "And it cannot wield a weapon");
```

В данном примере кода создается также экземпляр `ninja` и проверяется, относится ли он к классу `Ninja` и может ли представленный им ниндзя владеть оружием, как показано ниже. А поскольку каждый ниндзя — также человек, то дополнительно проверяется следующее: относится ли экземпляр `ninja` к типу `Person`, т.е. является также экземпляром данного типа, имеет ли ниндзя имя (`name`) и пользуется ли он танцем как составной частью своего боевого искусства.

```
var ninja = new Ninja("Yoshi", "Wakizashi");
assert(ninja instanceof Ninja, "A ninja's a ninja");
assert(ninja.wieldWeapon(), "That can wield a weapon");
assert(ninja instanceof Person, "But it's also a person");
assert(ninja.name === "Yoshi" , "That has a name");
assert(ninja.dance(), "And enjoys dancing");
```

Насколько описанные здесь нововведения в стандарт ES6 упрощают дело? Прежде всего, теперь не нужно думать о прототипах или побочных эффектах, вытекающих из некоторых переопределяемых свойств. Кроме того, при определении классов можно указывать их отношения наследования с помощью ключевого слова `extends`. И, наконец, многие разработчики, перешедшие на JavaScript из таких языков программирования, как Java или C#, могут быть вполне удовлетворены.

Вот, собственно, и все. Начиная со стандарта ES6, у разработчиков приложений на JavaScript появилась возможность создавать иерархии классов почти также просто, как и в любом другом, более традиционном объектно-ориентированном языке программирования.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- В языке JavaScript объекты являются простыми коллекциями именованных свойств с конкретными значениями.

- В языке JavaScript применяются прототипы.
- У каждого объекта имеется ссылка на *прототип* — объект, в котором выполняется поиск конкретного свойства, если оно отсутствует у самого объекта. У прототипа объекта, в свою очередь, может быть свой прототип и т.д., в результате чего образуется *цепочка прототипов*.
- Прототип объекта можно определить с помощью метода `Object.setPrototypeOf()`.
- Прототипы тесно связаны с функциями-конструкторами. У каждой такой функции имеется свойство `prototype`, где задается прототип тех объектов, экземпляры которых создаются с помощью конструктора.
- У объекта `prototype` функции имеется свойство `constructor`, указывающее обратно на саму функцию. Это свойство доступно всем объектам, экземпляры которых создаются с помощью данной функции. С помощью свойства `constructor` можно выяснить, хотя и с некоторыми ограничениями, был ли объект создан конкретной функцией.
- В языке JavaScript почти все можно изменить во время выполнения программы, включая и прототипы объектов и функций!
- Если требуется, чтобы экземпляры, полученные с помощью функции-конструктора объектов типа `Ninja`, наследовали, а точнее, имели доступ к свойствам, которые доступны экземплярам, полученным с помощью функции-конструктора объектов типа `Person`, в прототипе функции-конструктора объектов типа `Ninja` следует задать новый экземпляр класса `Person`.
- У свойств в языке JavaScript имеются атрибуты (`configurable`, `enumerable`, `writable` и прочие поля). Эти атрибуты могут быть определены с помощью встроенного метода `Object.defineProperty()`.
- В стандарт ES6 языка JavaScript введено ключевое слово `class`, упрощающее имитацию классов. Хотя внутренний механизм наследования классов по-прежнему основывается на прототипах!
- Ключевое слово `extends` позволяет более изящно реализовать наследование.

Упражнения

1. Какое из приведенных ниже свойств указывает на объект, поиск в котором будет осуществлен, если у целевого объекта отсутствует искомое свойство?

   а) `class`

   б) `instance`

   в) `prototype`

г) `pointTo`

2. Какое значение примет переменная `a1` после выполнения приведенного ниже фрагмента кода?

```
function Ninja(){}
  Ninja.prototype.talk = function (){
    return "Hello";
};

const ninja = new Ninja();
const a1 = ninja.talk();
```

3. Какое значение примет переменная `a1` после выполнения приведенного ниже фрагмента кода?

```
function Ninja(){}
Ninja.message = "Hello";

const ninja = new Ninja();

const a1 = ninja.message;
```

4. Поясните отличия в методе `getFullName()`, обнаруживаемые в двух приведенных ниже фрагментах кода.

```
// Первый фрагмент
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;

  this.getFullName = function () {
    return this.firstName + " " + this.lastName;
  }
}

// Второй фрагмент
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function () {
  return this.firstName + " " + this.lastName;
}
```

5. На что будет указывать свойство `ninja.constructor` после выполнения приведенного ниже фрагмента кода?

```
function Person() { }
function Ninja() { }
const ninja = new Ninja();
```

6. На что будет указывать свойство `ninja.constructor` после выполнения приведенного ниже фрагмента кода?

```
function Person() { }
function Ninja() { }
```

```
Ninja.prototype = new Person();
const ninja = new Ninja();
```

7. Поясните, каким образом операция `instanceof` действует в следующем примере кода:

```
function Warrior() { }

function Samurai() { }
Samurai.prototype = new Warrior();

var samurai = new Samurai();

samurai instanceof Warrior; // пояснить
```

8. Преобразуйте следующий фрагмент кода из стандарта ES6 в стандарт ES5:

```
class Warrior {
  constructor(weapon){
    this.weapon = weapon;
  }

  wield() {
    return "Wielding " + this.weapon;
  }

  static duel(warrior1, warrior2){
    return warrior1.wield() + " " + warrior2.wield();
  }
}
```

8 Управление доступом к объектам

В этой главе...

- Применение методов получения и установки для управления доступом к объектам
- Управление доступом к объектам через прокси-объекты
- Применение прокси-объектов для реализации сквозных функциональных возможностей

Как пояснялось в предыдущей главе, объекты в JavaScript являются динамическими коллекциями свойств. В них можно без особого труда добавлять новые свойства, изменять значения уже имеющихся свойств и даже полностью удалять существующие свойства. Во многих случаях (например, при проверке значения свойств, протоколировании или отображении данных в пользовательском интерфейсе) требуется строгий контроль над тем, что происходит с объектами. И в этой главе представлены способы и средства, применяемые для управления доступом к объектам и контроля над всеми происходящими в них изменениями.

Сначала в главе рассматриваются методы получения и установки, с помощью которых можно управлять доступом к отдельным свойствам объектов. Применение этих методов уже демонстрировалось в главах 5 и 7, а в этой главе будет показана их поддержка, встроенная в язык, и продемонстрировано их применение протоколирования, проверки достоверности вводимых данных и определения вычисляемых свойств.

Далее в этой главе рассматриваются прокси-объекты — совершенно новый тип объектов, веденный в стандарт ES6. Эти объекты служат для управления доступом к другим объектам. Из этой главы вы узнаете, каким образом действуют прокси-объекты, как с их помощью можно эффективно расширить прикладной код сквозными функциональными возможностями, включая измерение производительности и протоколирование, и как избежать исключений из-за неинициализированных данных, автоматически заполняя свойства объектов. Итак, начнем наше исследование с уже известных отчасти методов получения и установки.

Знаете ли вы?

Какие преимущества доступа к значениям свойств дают методы получения и установки?

В чем заключается главное отличие прокси-объектов от методов получения и установки?

Какие три вида скрытых препятствий таят в себе прокси-объекты?

8.1. Управление доступом к свойствам объектов с помощью методов получения и установки

Объекты в JavaScript представляют собой относительно простые коллекции свойств. Основной способ слежения за состоянием программы состоит в видоизменении этих свойств. Рассмотрим в качестве примера следующий фрагмент кода:

```
function Ninja (level) {
  this.skillLevel = level;
}
const ninja = new Ninja(100);
```

В приведенном выше фрагменте кода определяется функция-конструктор объектов типа `Ninja`, с помощью которой создается объект `ninja` со свойством `skillLevel`. И если в дальнейшем потребуется изменить значение данного свойства, то можно написать следующую строку кода:

```
ninja.skillLevel = 20
```

Все это, конечно, изящно и удобно, но что произойдет в следующих случаях, когда требуется:

- застраховаться от таких случайных ошибок, как присваивание непредвиденных значений, например, неверного типа:

  ```
  ninja.skillLevel = "high"
  ```

- регистрировать все изменения значения свойства `skillLevel`;

- отображать значение свойства `skillLevel` где-нибудь в пользовательском интерфейсе веб-страницы. Естественно, что при этом нужно получить самое последнее значение свойства, но как это проще всего сделать?

Методы получения и установки уже упоминались в главе 5 как средства имитации закрытых свойств объектов в JavaScript через замыкания. Рассмотрим еще раз, каким образом выполняется управление доступом к закрытым свойствам объектов с помощью методов получения и установки на примере кода, приведенного в листинге 8.1.

Листинг 8.1. Защита закрытых свойств с помощью методов получения и установки

```
function Ninja () {
  let skillLevel;
  this.getSkillLevel = () => skillLevel;

  this.setSkillLevel = value => {
    skillLevel = value;
  };
}

const ninja = new Ninja();
ninja.setSkillLevel(100);
assert(ninja.getSkillLevel() === 100,
       "Our ninja is at level 100!");
```

Определить закрытую переменную **skillLevel**

Метод получения управляет доступом к закрытой переменной **skillLevel**

Метод установки управляет значениями, которые могут быть присвоены переменной **skillLevel**

Задать новое значение переменной **skillLevel** через метод установки

Извлечь значение из переменной **skillLevel** с помощью метода получения

В данном примере кода определяется функция-конструктор объектов типа `Ninja`, где создаются объекты ниндзя с "закрытым" свойством (или переменной) `skillLevel`, доступным только через методы `getSkillLevel()` и `setSkillLevel()`. В частности, существующее значение этого свойства может быть получено только через метод `getSkillLevel()`, тогда как новое его значение установлено только через метод `setSkillLevel()` (вспомните о замыканиях, представленных в главе 5).

Если же потребуется зарегистрировать все попытки прочитать значение свойства `skillLevel`, то придется расширить метод `getSkillLevel()`. А если потребуется отреагировать на все попытки записать значение в данное свойство, то придется расширить метод `setSkillLevel()`, как показано в следующем фрагменте кода:

```
function Ninja () {
  let skillLevel;

  this.getSkillLevel = () => {
    report("Getting skill level value");
    return skillLevel;
  };

  this.setSkillLevel = value => {
```

С помощью метода получения можно постоянно контролировать доступ к свойству

```
    report("Modifying skillLevel property from:",
           skillLevel, "to: ", value);
    skillLevel = value;
  }
}
```

С помощью метода установки можно постоянно контролировать установку нового значения свойства

Эти методы примечательны тем, что они позволяют легко реагировать на все взаимодействия со свойствами. С их помощью можно, например, оперативно подключать протоколирование, проверку достоверности и прочие побочные эффекты вроде видоизменений в пользовательском интерфейсе.

Но в связи с этим может закрасться тревожная мысль. Ведь свойство `skillLevel` служит для хранения значений; в нем делается ссылка на данные (число **100**), а не на функцию. К сожалению, чтобы воспользоваться всеми преимуществами управляемого доступа, все взаимодействия со свойством придется осуществлять, вызывая явным образом соответствующие методы, что, откровенно говоря, не совсем удобно.

К счастью, в язык JavaScript встроена поддержка настоящих методов получения и установки свойств, доступных через обычные свойства данных (например, по ссылке `ninja.skillLevel`). Но, кроме того, эти методы способны вычислять значение запрашиваемого свойства, проверять достоверность переданного им значения и обрабатывать его должным образом. Поэтому рассмотрим эту встроенную поддержку методов получения и установки более подробно.

8.1.1. Определение методов получения и установки

В языке JavaScript методы получения и установки могут быть определены двумя способами:

- путем указания их в литералах объектов или в определениях классов, возможность создавать которые появилась в стандарте ES6;
- путем вызова встроенного метода `Object.defineProperty()`.

Явная поддержка методов получения и установки появилась еще в стандарте ES5. Исследуем синтаксис определения этих методов, как всегда, на конкретном примере. Так, в примере кода из листинга 8.2 определяется объект для хранения списка имен ниндзя и требуется получить и установить имя первого ниндзя в списке.

Листинг 8.2. Определение методов получения и установки в литералах объектов

```
const ninjaCollection = {
  ninjas: ["Yoshi", "Kuma", "Hattori"],
  get firstNinja(){
    report("Getting firstNinja");
    return this.ninjas[0];
  },
```

Определить для свойства `firstNinja` метод получения, возвращающий имя первого ниндзя из списка и выводящий соответствующее сообщение

```
  set firstNinja(value){
    report("Setting firstNinja");
    this.ninjas[0] = value;
  }
};

assert(ninjaCollection.firstNinja === "Yoshi",
       "Yoshi is the first ninja");

ninjaCollection.firstNinja = "Hachi";

assert(ninjaCollection.firstNinja === "Hachi"
       && ninjaCollection.ninjas[0] === "Hachi",
       "Now Hachi is the first ninja");
```

Определить для свойства `firstNinja` метод установки, изменяющий имя первого ниндзя в списке и выводящий соответствующее сообщение

Получить доступ к свойству `firstNinja` как к обычному свойству объекта

Изменить значение свойства `firstNinja` как обычное свойство объекта

Убедиться, что изменение свойства сохранено

В данном примере кода определяется объект `ninjaCollection`, содержащий обычное свойство `ninjas`, ссылающееся на массив имен ниндзя, а также методы получения и установки свойства `firstNinja`. Общий синтаксис определения методов получения и установки наглядно показан на рис. 8.1.

Как видите, в определении метода получения имя свойства предваряется ключевым словом `get`, а в определении метода установки — ключевым словом `set`. В коде из листинга 8.2 методы получения и установки выводят соответствующие сообщения. Кроме того, метод получения возвращает имя ниндзя по индексу `0`, а метод установки присваивает новое имя ниндзя по тому же самому индексу:

```
get firstNinja(){
  report("Getting firstNinja");
  return this.ninjas[0];
},
set firstNinja(value){
  report("Setting firstNinja");
  this.ninjas[0] = value;
}
```

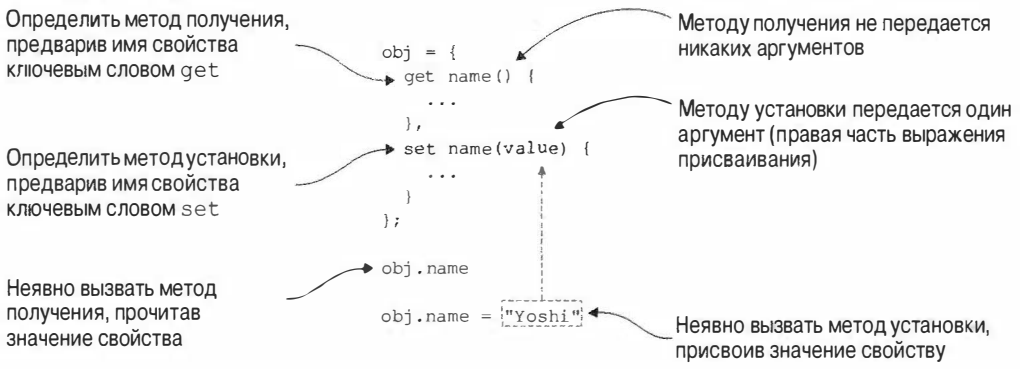


Рис. 8.1. Синтаксис определения методов получения и установки. При этом имя свойства предваряется ключевым словом **`get`** или **`set`**

Далее в рассматриваемом здесь примере кода проверяется, что в результате доступа к свойству `firstNinja` метод получения возвращает имя первого ниндзя (`Yoshi`):

```
assert(ninjaCollection.firstNinja === "Yoshi",
       "Yoshi is the first ninja");
```

Обратите внимание на то, что доступ к данному свойству формально осуществляется как к обычному свойству объекта, а не через метод получения, как это есть на самом деле. При доступе к свойству `firstNinja` неявно вызывается метод получения, выводящий сообщение `"Getting firstNinja"` (Получение значения свойства `firstNinja`) и возвращающий имя ниндзя по индексу **0**.

После этого с помощью метода установки свойству `firstNinja` присваивается новое значение. И в этом случае новое значение формально присваивается таким же образом, как и обычному свойству объекта:

```
ninjaCollection.firstNinja = "Hachi";
```

Как и в предыдущем случае, для записи нового значения свойства `firstNinja` неявно вызывается метод установки, который выводит сообщение `"Setting firstNinja"` (Установка свойства `firstNinja`) и видоизменяет имя ниндзя по индексу **0**.

И, наконец, в анализируем здесь примере кода проверяется результат изменения значения свойства и наличие нового имени ниндзя по индексу **0**, которое может быть доступно как явно из массива `ninjas`, так и неявно через метод получения, как показано ниже.

```
assert(ninjaCollection.firstNinja === "Hachi"
       && ninjaCollection.ninjas[0] === "Hachi",
       "Now Hachi is the first ninja");
```

Результат выполнения кода из листинга 8.2 приведен на рис. 8.2. При доступе к свойству `firstNinja` (например, по ссылке `ninjaCollection.firstNinja`) сразу же вызывается метод получения, который в данном случае выводит сообщение `"Getting firstNinja"`. Далее проверяется результат обращения к данному свойству (получено имя `Yoshi`) и выводится сообщение `"Yoshi is the first ninja"` (Йоси — первый ниндзя). После этого свойству `firstNinja` присваивается новое значение, для чего неявно вызывается метод установки, который выводит сообщение `"Setting firstNinja"`.

Из рассмотренного выше примера можно сделать следующий важный вывод: встроенные методы получения и установки позволяют сделать доступ к свойствам привычным нам образом. Эти методы вызываются неявно при доступе к свойству. И это еще раз наглядно показано на рис. 8.3.

Рассмотренный выше синтаксис определения методов получения и установки довольно прост, и поэтому не удивительно, что разработчики веб-приложений на JavaScript часто пользуются им и в других случаях. А в следующем примере кода из листинга 8.3 для определения методов получения и установки используются классы, которые стали доступны в стандарте ES6.

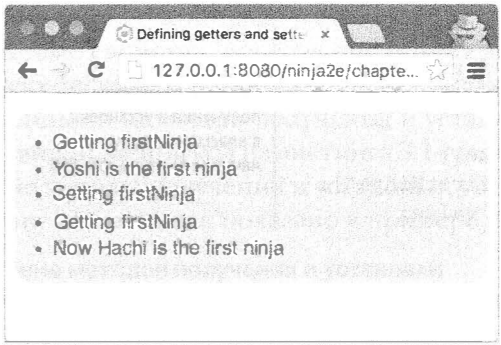
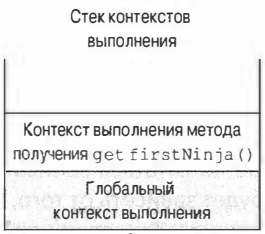


Рис. 8.2. Результат выполнения кода из листинга 8.2. Если у свойства имеются методы получения и установки, то первый из них неявно вызывается всякий раз, когда считывается значение этого свойства, а второй — когда данному свойству присваивается новое значение

```
const ninjaCollection = {
    ninjas: ["Yoshi", "Kuma", "Hattori"],
    get firstNinja() {
        report("Getting firstNinja");
        return this.ninjas[0];
    },
    ...
};

assert(ninjaCollection.firstNinja === "Yoshi",
    "Yoshi is the first ninja");
```



При доступе к свойству `firstNinja` сразу же запускается соответствующий метод получения

Когда вызывается метод получения, в стеке создается и размещается соответствующий контекст выполнения. Этот процесс аналогичен вызову обычной функции

Рис. 8.3. При доступе к свойству неявно вызывается его встроенный метод получения. Данный процесс отличается от вызова обычного метода лишь тем, что все происходит "за кадром". Аналогичный процесс происходит и при присваивании свойству нового значения через неявно вызываемый метод установки

Код из данного примера является видоизмененным вариантом кода из листинга 8.2 и включает в себя классы, которые стали доступными в стандарте ES6. В нем сохранены тесты, в которых проверяется, действует ли код по-прежнему именно так, как и предполагалось.

Листинг 8.3. Определение методов получения и установки с помощью классов в стандарте ES6

```
class NinjaCollection {
  constructor(){
    this.ninjas = ["Yoshi", "Kuma", "Hattori"];
  }
```

```
  get firstNinja(){
    report("Getting firstNinja");
    return this.ninjas[0];
  }
  set firstNinja(value){
    report("Setting firstNinja");
    this.ninjas[0] = value;
  }
}
const ninjaCollection = new NinjaCollection();

assert(ninjaCollection.firstNinja === "Yoshi",
       "Yoshi is the first ninja");

ninjaCollection.firstNinja = "Hachi";

assert(ninjaCollection.firstNinja === "Hachi"
       && ninjaCollection.ninjas[0] === "Hachi",
       "Now Hachi is the first ninja");
```

Определить методы получения и установки в классе, что допустимо, начиная со стандарта ES6

Примечание

Для отдельного свойства не всегда требуется определять методы получения и установки. Зачастую достаточно, например, определить метод получения. И если в этом случае попытаться записать значение в данное свойство, то конкретное поведение будет зависеть от того, выполняется ли прикладной код в строгом или нестрогом режиме. Так, если код выполняется в нестрогом режиме, присваивание значения свойству, наделенному только методом получения, ни к чему не приведет, а интерпретатор JavaScript негласно проигнорирует запрос. А если код выполняется в строгом режиме, то интерпретатор JavaScript сгенерирует ошибку несоответствия типов, указывающую на попытку присвоить значение свойству, наделенному методом получения, но не методом установки.

Несмотря на то что определить методы получения и установки с помощью литералов объектов и классов, ставших доступными в стандарте ES6, совсем не трудно, обоим способам, как вы, возможно, заметили, все же чего-то недостает. По традиции методы получения и установки служат для управления доступом к закрытым свойствам объектов, как было показано в коде из листинга 8.1. К сожалению, в языке JavaScript не поддерживаются закрытые свойства объектов (см. главу 5). Конечно, можно попытаться сымитировать их через замыкания, определив переменные и методы объектов, которые охватят эти переменные, но добиться этого все же не удастся. Ведь методы получения и установки, определяемые с помощью литералов объектов и классов, доступных, начиная со стандарта ES6, не создаются в той же области видимости функции, где и переменные, которые можно было бы использовать для имитации закрытых свойств объектов. К счастью, имеется другой способ — воспользоваться встроенным методом `Object.defineProperty()`.

Как пояснялось в главе 7, с помощью метода `Object.defineProperty()` можно определить новые свойства, передав ему объект-дескриптор свойства. Среди прочего, объект-дескриптор свойства может включать в себя свойства `get` и `set` с определенными методами получения и установки. Воспользуемся этой возможностью, видоизменив код из листинга 8.1 таким образом, чтобы реализовать встроенные методы получения и установки, управляющие доступом к "закрытому" свойству объекта, как показано в примере кода из листинга 8.4.

Листинг 8.4. Определение методов получения и установки с помощью метода `Object.defineProperty()`

```
function Ninja() {
  let _skillLevel = 0;

  Object.defineProperty(this, 'skillLevel', {
    get: () => {
      report("The get method is called");
      return _skillLevel;
    },
    set: value => {
      report("The set method is called");
      _skillLevel = value;
    }
  });
}

const ninja = new Ninja();

assert(typeof ninja._skillLevel === "undefined",
       "We cannot access a 'private' property");
assert(ninja.skillLevel === 0, "The getter works fine!");

ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");
```

Определить функцию-конструктор

Определить "закрытую" переменную, которая станет доступной через замыкания

Вызвать встроенный метод `Object.defineProperty()`, чтобы определить свойство `skillLevel`

Метод `get()`, вызываемый всякий раз, когда считывается значение свойства `skillLevel`

Метод `set()`, вызываемый всякий раз, когда свойству `skillLevel` присваивается значение

Создать новый объект типа `Ninja`

Закрытая переменная доступна не напрямую, а через метод `get()` свойства `skillLevel`

Метод `set()` вызывается неявно, когда свойству `skillLevel` присваивается значение

В данном примере кода сначала определяется функция-конструктор объектов типа `Ninja` с переменной `_skillLevel`, которую предполагается использовать в качестве закрытой переменной, как в коде из листинга 8.1. Затем с помощью встроенного метода `Object.defineProperty()` для вновь созданного объекта, доступного по ссылке `this`, определяется свойство `skillLevel`:

```
Object.defineProperty(this, 'skillLevel', {
  get: () => {
    report("The get method is called");
    return _skillLevel;
  },
```

```
  set: value => {
    report("The set method is called");
    _skillLevel = value;
  }
});
```

В данном случае свойство `skillLevel` требуется для управления доступом к закрытой переменной. Поэтому далее определяются методы `get()` и `set()`, которые будут вызываться всякий раз, когда осуществляется доступ к данному свойству.

Но в отличие от методов получения и установки, определяемых с помощью литералов объектов и классов, методы `get()` и `set()`, определяемые с помощью встроенного метода `Object.defineProperty()`, создаются в той же области видимости, где и "закрытая" переменная `skillLevel`. Оба метода образуют замыкание вокруг "закрытой" переменной, которая может быть доступна только через эти методы.

Остальная часть кода из рассматриваемого здесь примера действует таким же образом, как и в предыдущих примерах. В частности, создается новый объект типа `Ninja` и проверяется, что закрытая переменная недоступна непосредственно. Все взаимодействия проходят через методы получения и установки, но так, как будто речь идет об обычных свойствах объектов:

```
ninja.skillLevel === 0  <-- Активизировать метод получения
ninja.skillLevel = 10   <-- Активизировать метод установки
```

Как видите, способ определения методов получения и установки с помощью метода `Object.defineProperty()` оказывается более многословным и сложным, чем с помощью литералов объектов и классов. Но в некоторых случаях, когда требуются закрытые свойства объектов, он вполне пригоден.

Независимо от способа определения, методы получения и установки позволяют задавать свойства объектов, которые можно использовать привычным нам образом, хотя в этих методах всякий раз можно выполнять дополнительный код, когда осуществляется чтение или запись значения конкретного свойства. И эти средства особенно удобны для организации протоколирования, проверки достоверности присваиваемых значений и даже извещения других частей кода о происходящих изменениях. Исследуем некоторые примеры применения подобных методов.

8.1.2. Применение методов получения и установки для проверки достоверности значений свойств

Как утверждалось ранее, метод установки запускается всякий раз, когда требуется изменить значение соответствующего свойства. Методами установки выгодно пользоваться для выполнения определенного действия всякий раз, когда в прикладном коде предпринимается попытка обновить значение свойства. Например, можно проверить достоверность значения, переданного методу установки. В качестве примера рассмотрим код из листинга 8.5, где прове-

ряется возможность присваивать свойству skillLevel только целочисленные значения.

Листинг 8.5. Проверка достоверности значений, присваиваемых свойствам, в методах установки

```
function Ninja() {
  let _skillLevel = 0;

  Object.defineProperty(this, 'skillLevel', {
    get: () => _skillLevel,
    set: value => {
      if(!Number.isInteger(value)){
        throw new TypeError("Skill level should be a number");
      }
      _skillLevel = value;
    }
  });
}

const ninja = new Ninja();

ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");

try {
  ninja.skillLevel = "Great";
  fail("Should not be here");
} catch(e){
  pass("Setting a non-integer value throws an exception");
}
```

Проверить, является ли переданное значение целочисленным. Если оно таковым не является, сгенерировать исключение

Свойству можно присвоить целочисленное значение

Попытка присвоить нецелочисленное значение (в данном случае строковое) приведет к исключению, генерируемому в методе установки

Код в данном примере служит прямым расширением кода из листинга 8.4. А отличается он лишь тем, что всякий раз, когда свойству skillLevel присваивается значение, в методе установки проверяется, является ли переданное ему значение целочисленным. Если оно таковым не является, то генерируется исключение и значение закрытой переменной _skillLevel не меняется. А если проверка пройдет нормально, то новое целочисленное значение присваивается закрытой переменной _skillLevel, как показано ниже.

```
set: value => {
  if(!Number.isInteger(value)){
    throw new TypeError("Skill level should be a number");
  }
  _skillLevel = value;
}
```

При тестировании кода в данном примере сначала проверяется, нормально ли выполнено присваивание целочисленного значения:

```
ninja.skillLevel = 10;
assert(ninja.skillLevel === 10, "The value was updated");
```

А затем проверяется ситуация, в которой свойству ошибочно присваивается значение другого типа, в частности, строковое. В таком случае должно быть сгенерировано исключение, как следует из приведенного ниже фрагмента кода.

```
try {
  ninja.skillLevel = "Great";
  fail("Should not be here");
} catch(e){
  pass("Setting a non-integer value throws an exception");
}
```

Именно так исключаются все нелепые мелкие программные ошибки, которые возникают в том случае, если определенному свойству присваивается значение неверного типа. Это, безусловно, требует дополнительных издержек, но это именно та цена, которую приходится платить за безопасное пользование таким высокодинамичным языком, как JavaScript.

Это лишь один из многих примеров полезного применения методов установки. По такому же принципу можно, например, отслеживать предысторию значений, выполнять протоколирование, уведомлять о внесенных изменениях и делать многое другое.

8.1.3. Применение методов получения и установки для определения вычисляемых свойств

Помимо возможности управлять доступом к определенным свойствам объектов, методы получения и установки можно применять для определения *вычисляемых свойств*, т.е. таких свойств, значения которых вычисляются по запросу. Значения вычисляемых свойств нигде не хранятся. Вместо этого для них создаются методы `get()` и/или `set()`, которые позволяют косвенным образом получить или установить значения на основе значений других свойств. В примере кода из листинга 8.6 определяется объект с двумя свойствами, `name` и `clan`, которые служат для вычисления свойства `fullTitle`.

Листинг 8.6. Определение вычисляемых свойств

```
const shogun = {
  name: "Yoshiaki",
  clan: "Ashikaga",
  get fullTitle(){
    return this.name + " " + this.clan;
  },
  set fullTitle(value) {
    const segments = value.split(" ");
    this.name = segments[0];
    this.clan = segments[1];
  }
};
```

Определить в литерале объекта метод получения для свойства `fullTitle`, значение которого вычисляется путем сцепления строковых значений двух других свойств объекта

Определить в литерале объекта метод установки для свойства `fullTitle`, где разделяется переданное значение и обновляются значения двух других обычных свойств объекта

```
assert(shogun.name === "Yoshiaki", "Our shogun Yoshiaki");
assert(shogun.clan === "Ashikaga", "Of clan Ashikaga");
assert(shogun.fullTitle === "Yoshiaki Ashikaga",
       "The full name is now Yoshiaki Ashikaga");

shogun.fullTitle = "Ieyasu Tokugawa";
assert(shogun.name === "Ieyasu", "Our shogun Ieyasu");
assert(shogun.clan === "Tokugawa", "Of clan Tokugawa");
assert(shogun.fullTitle === "Ieyasu Tokugawa",
       "The full name is now Ieyasu Tokugawa");
```

Свойства `name` и `clan` являются обычными свойствами, значения которых получаются непосредственно. А при доступе к свойству `fullTitle` вызывается метод `get()`, который вычисляет значение этого свойства

Присваивание значения свойству `fullTitle` приводит к вызову метода `set()`, где вычисляются новые значения, которые затем присваиваются свойствам `name` и `clan`

В данном примере кода определяется объект `shogun` с двумя стандартными свойствами, `name` и `clan`, а также методы получения и установки значения вычисляемого свойства `fullTitle`:

```
const shogun = {
  name: "Yoshiaki",
  clan: "Ashikaga",
  get fullTitle(){
    return this.name + " " + this.clan;
  },
  set fullTitle(value) {
    const segments = value.split(" ");
    this.name = segments[0];
    this.clan = segments[1];
  }
};
```

Значение свойства `fullTitle` вычисляется в методе `get()` по запросу путем сцепления строковых значений свойств `name` и `clan`. А в методе `set()` вызывается встроенный метод `split()`, доступный для всех текстовых строк и предназначенный для разделения строк на сегменты по символу пробела. Первый сегмент представляет имя, присваиваемое свойству `name`, а второй сегмент — клан, присваиваемый свойству `clan`.

Таким образом, код из рассматриваемого здесь примера отвечает сразу за оба направления: вычисление значения свойства `fullTitle` при его чтении и видоизменение свойств, из значений которых составляется значение свойства `fullTitle` при его записи.

Откровенно говоря, мы не склонны пользоваться вычисляемыми свойствами. Столь же полезным мог бы оказаться и метод `getFullTitle()`, хотя вычисляемые свойства могут принципиально повысить удобочитаемость исходного кода. Если определенное значение (в данном случае значение свойства `fullTitle`) зависит *только* от внутреннего состояния объекта (в данном случае от свойств `name` и `clan`), то его лучше представить в виде поля данных (или свойства), а не функции.

На этом исследование методов получения и установки завершается. Как видите, они являются удобным дополнением языка и помогают реализовать протоколирование, проверку достоверности данных и обнаружение изменений в значениях свойств. Но этого, к сожалению, не всегда достаточно. Ведь иногда требуется управлять всеми видами взаимодействий с объектами, и для этой цели можно воспользоваться *прокси-объектами* — совершенно новым типом объектов, введенным в стандарт ES6.

8.2. Применение прокси-объектов для управления доступом

Прокси-объект — это такой заменитель объекта, через который можно управлять доступом к другому объекту. Он дает возможность определить специальные действия, которые будут выполняться при взаимодействии с объектом, например, при получении или установке значения свойства или вызове метода. Прокси-объекты можно рассматривать в качестве обобщения методов получения и установки. Но если с помощью каждого метода получения и установки можно управлять доступом только к одному свойству объекта, то прокси-объекты позволяют вообще управлять всеми взаимодействиями с объектом, включая и вызовы методов.

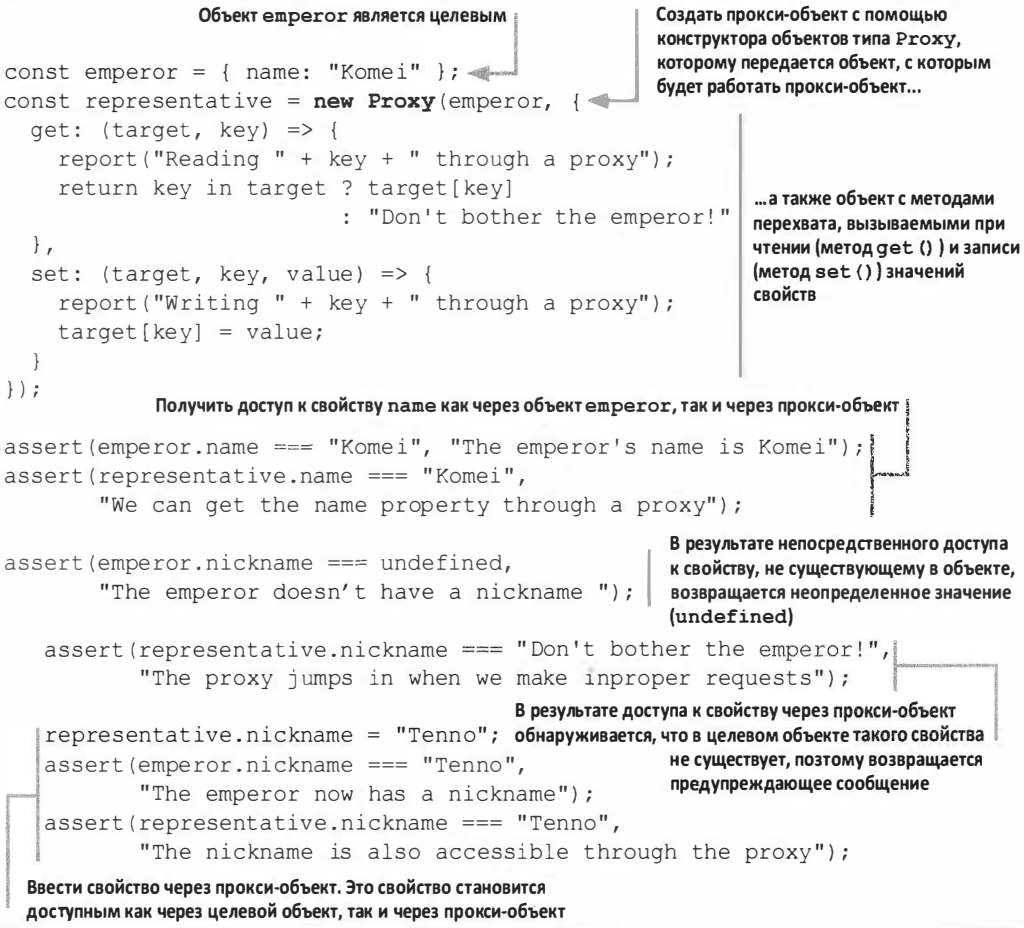
Прокси-объекты можно применять там, где по традиции используются методы получения и установки, например, для протоколирования, проверки достоверности данных и реализации вычисляемых свойств. Но этим потенциальные возможности прокси-объектов не исчерпываются. С их помощью нетрудно ввести профилирование и измерение производительности в прикладной код, автоматически заполнять свойства объектов, чтобы избежать неприятных исключений из-за неинициализированных свойств, а также заключить в оболочку объекты исполняющей среды (например, модели DOM), чтобы свести к минимуму несовместимость браузеров.

Примечание

Прокси-объекты введены в стандарт ES6. О текущей их поддержке в браузерах можно узнать по адресу `http://kangax.github.io/compat-table/es6/#test-Proxy`.

В языке JavaScript прокси-объекты можно создавать с помощью встроенного конструктора объектов типа `Proxy`. Рассмотрим сначала простой пример прокси-объекта, перехватывающего все попытки прочитать и записать значения свойства объекта, как демонстрируется в коде из листинга 8.7.

Листинг 8.7. Создание прокси-объектов с помощью конструктора объектов типа `Proxy`




```
const emperor = { name: "Komei" };
const representative = new Proxy(emperor, {
  get: (target, key) => {
    report("Reading " + key + " through a proxy");
    return key in target ? target[key]
                         : "Don't bother the emperor!"
  },
  set: (target, key, value) => {
    report("Writing " + key + " through a proxy");
    target[key] = value;
  }
});

assert(emperor.name === "Komei", "The emperor's name is Komei");
assert(representative.name === "Komei",
       "We can get the name property through a proxy");

assert(emperor.nickname === undefined,
       "The emperor doesn't have a nickname ");

  assert(representative.nickname === "Don't bother the emperor!",
         "The proxy jumps in when we make inproper requests");

  representative.nickname = "Tenno";
  assert(emperor.nickname === "Tenno",
         "The emperor now has a nickname");
  assert(representative.nickname === "Tenno",
         "The nickname is also accessible through the proxy");
```

Сначала в данном примере кода создается базовый объект `emperor` с единственным свойством `name`. Затем с помощью встроенного конструктора объектов типа `Proxy` объект `emperor`, обычно называемый *целевым*, заключается в оболочку прокси-объекта `representative`. При создании прокси-объекта в качестве второго аргумента конструктору передается объект, в котором определяются методы *перехвата*, вызываемые при выполнении определенных действий над объектом:

```
const representative = new Proxy(emperor, {
  get: (target, key) => {
    report("Reading " + key + " through a proxy");
    return key in target ? target[key]
                         : "Don't bother the emperor!"
  },
  set: (target, key, value) => {
```

```
    report("Writing " + key + " through a proxy");
    target[key] = value;
  }
});
```

В данном случае определяются следующие два метода перехвата: метод `get()`, вызываемый при любой попытке получить значение свойства через прокси-объект, а также метод `set()`, вызываемый всякий раз, когда значение свойства устанавливается через прокси-объект. Как показано ниже, метод перехвата `get()` выполняет следующие действия: если у целевого объекта имеется искомое свойство, оно возвращается, а если это свойство отсутствует, то возвращается сообщение, предупреждающее пользователя не беспокоить по пустякам императора, представленного объектом `emperor`.

```
get: (target, key) => {
  report("Reading " + key + " through a proxy");
  return key in target ? target[key]
                       : "Don't bother the emperor!"
}
```

Далее в рассматриваемом здесь примере кода проверяется возможность доступа к свойству `name` как непосредственно через целевой объект `emperor`, так и через прокси-объект, как показано в следующем фрагменте кода:

```
assert(emperor.name === "Komei", "The emperor's name is Komei");
assert(representative.name === "Komei",
       "We can get the name property through a proxy");
```

Если доступ к свойству `name` осуществляется непосредственно через объект `emperor`, то возвращается строковое значение `"Komei"`. Но если доступ к данному свойству осуществляется через прокси-объект `representative`, то неявно вызывается метод перехвата `get()`. А поскольку свойство `name` находится в целевом объекте `emperor`, то возвращается строковое значение `"Komei"`, как наглядно показано на рис. 8.4.

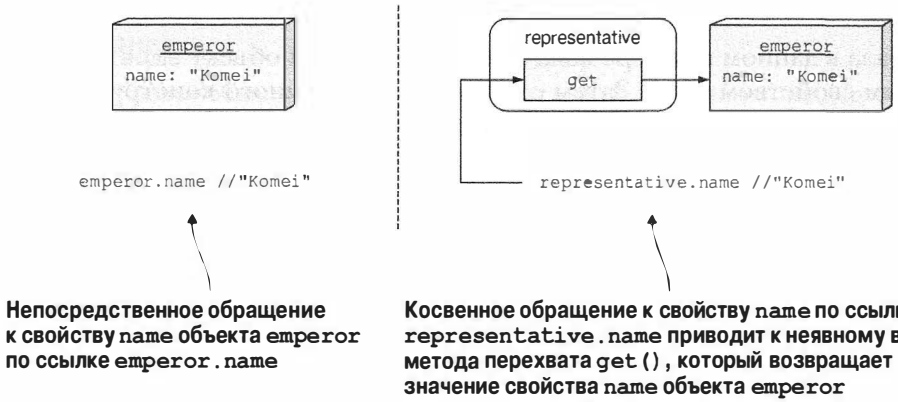


Рис. 8.4. Непосредственный (*слева*) и косвенный (*справа*) доступ к свойству `name` через прокси-объект

Примечание

Следует особенно подчеркнуть, что методы перехвата активизируются в прокси-объекте таким же образом, как и методы получения и установки в обычном объекте. Как только начнет выполняться действие (например, доступ к значению свойства через прокси-объект), неявно вызывается соответствующий метод перехвата, а интерпретатор JavaScript выполняет такие же действия, как и при явном вызове функции.

А если непосредственно обратиться к свойству `nickname`, несуществующему в целевом объекте `emperor`, то, как и следовало ожидать, будет получено неопределенное значение (`undefined`). Но если попытаться получить доступ к данному свойству через прокси-объект, то будет активизирована обработка ошибки в методе перехвата `get()` непосредственно в прокси-объекте. А поскольку у целевого объекта `emperor` отсутствует свойство `nickname`, то метод перехвата `get()` возвратит сообщение `"Don't bother the emperor!"` (Не беспокоить императора!).

Далее в анализируемом здесь примере кода новому свойству прокси-объекта присваивается значение:

```
representative.nickname = "Tenno"
```

Это присваивание выполняется через прокси-объект, а не напрямую, и поэтому вызывается метод перехвата `set()`, где выводится сообщение и присваивается значение свойству целевого объекта `emperor`:

```
set: (target, key, value) => {
  report("Writing " + key + " through a proxy");
  target[key] = value;
}
```

Естественно, что вновь созданное свойство может быть доступно как через прокси-объект, так и через целевой объект следующим образом:

```
assert(emperor.nickname === "Tenno",
       "The emperor now has a nickname");
assert(representative.nickname === "Tenno",
       "The nickname is also accessible through the proxy");
```

Суть применения прокси-объектов заключается в следующем: с помощью конструктора объекта типа `Proxy` создается прокси-объект, управляющий доступом к целевому объекту, активизируя определенные методы перехвата всякий раз, когда операция выполняется непосредственно над прокси-объектом.

В данном примере применяются методы перехвата `get()` и `set()`, но имеется немало других встроенных методов перехвата, позволяющих определять обработчики различных действий над объектами (подробнее см. по адресу `https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Proxy`). Ниже приведены некоторые примеры таких методов.

- Метод перехвата `apply()`, активизируемый при вызове функции, а метод перехвата `construct()` — при выполнении операции `new`.

- Методы перехвата `get()` и `set()`, активизируемые при чтении и записи значения свойства.
- Метод перехвата `enumerate()`, активизируемый в цикле `for-in`.
- Методы перехвата `getPrototypeOf()` и `setPrototypeOf()`, активизируемые при получении и установке значения прототипа.

Перехватывать можно многие операции, но эта тема необъятна и выходит за рамки данной книги. Поэтому уделим внимание операциям, которые нельзя переопределить: равенства (**==** или **===**), **instanceof** и **typeof**.

Например, в выражении `x == y` (либо в более строгом, `x === y`) проверяется, ссылаются ли переменные `x` и `y` на одинаковые объекты (или на одно и то же значение). Данная операция равенства предполагает некоторые допущения. В частности, в результате сравнения двух объектов должно всегда возвращаться одно и то же значение из двух одинаковых объектов, чего нельзя гарантировать, если это значение определяется в функции, задаваемой пользователем. Кроме того, само сравнение двух объектов не должно предоставлять доступ к одному из этих объектов, что могло бы произойти, если бы можно было перехватить операцию равенства. По той самой же причине нельзя перехватить операции `instanceof` и `typeof`.

Итак, выяснив принцип действия прокси-объектов и порядок их создания, перейдем к исследованию некоторых особенностей их практического применения, в том числе для протоколирования, измерения производительности, автоматического заполнения свойств объектов и реализации массивов, доступ к элементам которых выполняется по отрицательным индексам. И начнем мы с протоколирования.

8.2.1. Применение прокси-объектов для протоколирования

Одним из самых эффективных средств, позволяющих выяснить, как работает прикладной код и где искать источник неприятной программной ошибки, служит *протоколирование* – действие, направленное на вывод информации, которая может оказаться полезной в конкретный момент. С помощью протоколируемой информации можно, например, выяснить, какие именно функции вызывались, как долго они выполнялись, какие свойства читались или записывались и т.д.

К сожалению, когда используется режим протоколирования, то операторы вывода оказываются разбросанными по всему исходному коду. Обратимся к рассмотренному ранее примеру кода из листинга 8.4, немного изменив его, как показано в листинге 8.8.

Листинг 8.8. Протоколирование без прокси-объектов

```
function Ninja() {
  let _skillLevel = 0;

  Object.defineProperty(this, 'skillLevel', {
```

```
    get: () => {
      report("skillLevel get method is called");
      return _skillLevel;
    },
    set: value => {
      report("skillLevel set method is called");
      _skillLevel = value;
    }
  });
}

const ninja = new Ninja();
ninja.skillLevel;
ninja.skillLevel = 4;
```

Протоколирование осуществляется всякий раз, когда читается значение свойства `skillLevel`...

...и когда записывается значение в свойство `skillLevel`

Прочитать значение свойства `skillLevel` и активизировать метод `get()`

Записать значение в свойство `skillLevel` и активизировать метод `set()`

В данном примере кода определяется функция-конструктор объектов типа `Ninja`. В ней дополнительно определяются методы получения и установки значения свойства `skillLevel`, в которых фиксируются все попытки чтения и записи значения данного свойства.

Однако такое решение – далеко не идеально. Ведь прикладной код, выполняющий чтение и запись значения свойства, нам пришлось дополнить операторами протоколирования. И если в дальнейшем в объект `ninja` потребуется ввести дополнительные свойства, то придется проявить особое внимание, чтобы не забыть ввести и операторы протоколирования для каждого нового свойства.

К счастью, с помощью прокси-объектов можно активизировать протоколирование всякий раз, когда читается или записывается значение свойства, причем сделать это можно намного более ясно и изящно. В качестве примера рассмотрим код из листинга 8.9.

Листинг 8.9. Прокси-объекты упрощают внедрение протоколирования в объекты

```
function makeLoggable(target){
  return new Proxy(target, {
    get: (target, property) => {
      report("Reading " + property);
      return target[property];
    },

    set: (target, property, value) => {
      report("Writing value " + value + " to " + property);
      target[property] = value;
    }
  });
}
```

Определить функцию, которой передается целевой объект, а она задает для него режим протоколирования

Создать новый прокси-объект с целевым объектом

Метод перехвата `get()`, протоколирующий чтение значения свойства

Метод перехвата `set()`, протоколирующий запись значения свойства

```
let ninja = { name: "Yoshi"};
ninja = makeLoggable(ninja);

assert(ninja.name === "Yoshi", "Our ninja Yoshi");
ninja.weapon = "sword";
```

Создать новый объект ninja, предназначенный в качестве целевого, и включить для него режим протоколирования

Операции чтения и записи значения свойств в объекте протоколируются методами перехвата в прокси-объекте

В данном примере кода определяется функция `makeLoggable()`, которой передается целевой объект `target`, а она возвращает новый объект типа `Proxy`, обрабатывающий перехватываемые действия с помощью методов `get()` и `set()`. Помимо чтения и записи значения свойства, эти методы перехвата протоколируют информацию о свойстве, где читается и записывается значение.

Далее в рассматриваемом здесь примере кода создается объект `ninja` со свойством `name`. Затем он передается функции `makeLoggable()`, где он используется в качестве целевого объекта для вновь созданного прокси-объекта, который после этого присваивается обратно переменной `ninja`, тем самым переопределяя исходный объект. (Не волнуйтесь, исходный объект `ninja` никуда не девается, он по-прежнему поддерживается в активном состоянии в качестве целевого для прокси-объекта.)

Всякий раз, когда предпринимается попытка прочитать значение свойства (например, по ссылке `ninja.name`), вызывается метод перехвата `get()` и протоколируется информация о свойстве, значение которого было прочитано. Аналогичный процесс происходит и при записи значения свойства, как показано в следующей строке кода:

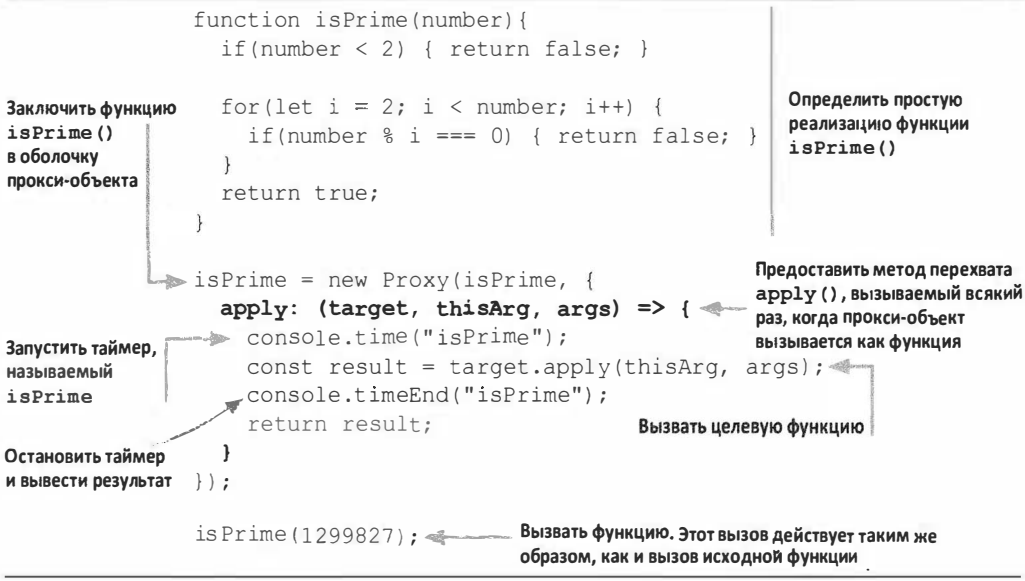
```
ninja.weapon = "sword"
```

Обратите внимание, насколько более простым и ясным оказывается такой способ по сравнению со стандартным применением методов получения и установки. В этом случае не приходится смешивать прикладной код с протоколирующим кодом и не нужно вводить дополнительные операторы протоколирования для каждого свойства объекта. Вместо этого все операции чтения и записи значений в свойствах проходят через методы перехвата в прокси-объекте. Протоколирование указывается лишь в одном месте и повторно используется столько раз и для стольких объектов, сколько потребуется.

8.2.2. Применение прокси-объектов для измерения производительности

Прокси-объекты можно применять не только для фиксации доступа к свойствам объектов, но и для измерения скорости работы вызываемых функций без изменения исходного кода самих функций. Допустим, требуется измерить производительность функции, определяющей, является ли число простым, как демонстрируется в примере кода из листинга 8.10.

Листинг 8.10. Измерение производительности с помощью прокси-объектов



В данном примере кода определяется простая функция `isPrime()`. (Конкретная функция не имеет особого значения, поэтому в качестве примера выбрана функция, выполнение которой может длиться заметное время.)

А теперь допустим, что требуется измерить скорость работы функции `isPrime()`, не внося изменения в ее исходный код. Эту функцию можно заключить в оболочку прокси-объекта, у которого имеется метод перехвата, вызываемый всякий раз, когда вызывается данная функция:

```
isPrime = new Proxy(isPrime, {
  apply: (target, thisArg, args) => {
    ...
  }
});
```

Функция `isPrime()` используется в качестве целевого объекта для вновь созданного прокси-объекта. Кроме того, прокси-объект снабжается методом-перехватчиком `apply()`, который будет запускаться всякий раз при вызове данной функции.

Как и в предыдущем примере кода, вновь созданный прокси-объект присваивается переменной `isPrime`. Благодаря этому отпадает необходимость вносить какие-либо изменения в код, вызывающий функцию, время выполнения которой измеряется. А для остальной части кода вносимые изменения совершенно незаметны. (Как это похоже на скрытные действия ниндзя!)

Всякий раз, когда вызывается функция `isPrime()`, ее вызов переадресуется методу перехвата `apply()` в прокси-объекте. В этом методе запускается таймер с помощью встроенного метода `console.time()` (см. главу 1), вызывается

исходная функция `isPrime()`, фиксируется истекшее время и возвращается результат вызова функции `isPrime()`.

8.2.3. Применение прокси-объектов для автоматического заполнения свойств

Помимо упрощения протоколирования, прокси-объекты можно применять для автоматического заполнения свойств объектов. Допустим, имеется модель структуры папок на компьютере, где у объекта папки могут быть свойства, которые также могут быть папками. А теперь допустим, что требуется смоделировать файл в конце длинного пути, как в следующем примере кода:

```
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

Чтобы добиться поставленной цели, можно было бы написать код, аналогичный приведенному ниже.

```
const rootFolder = new Folder();
rootFolder.ninjasDir = new Folder();
rootFolder.ninjasDir.firstNinjaDir = new Folder();
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

Для написания такого кода, по-видимому, потребуется больше труда, чем нужно, не так ли? Именно здесь и пригодится автоматическое заполнение свойств объектов. В качестве примера рассмотрим код из листинга 8.11.

Листинг 8.11. Автоматическое заполнение свойств объектов с помощью прокси-объектов

```
function Folder() {
  return new Proxy({}, {
    get: (target, property) => {
      report("Reading " + property);

      if(!(property in target)) {
        target[property] = new Folder();
      }
      return target[property];
    }
  });
}

const rootFolder = new Folder();

try {
  rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
  pass("An exception wasn't raised");
}
catch(e){
  fail("An exception has occurred");
}
```

Протоколировать все операции чтения данных из объекта

Если осуществляется доступ к несуществующему свойству, оно создается

Всякий раз, когда осуществляется доступ к свойству, активизируется метод перехвата `get()`, где создается свойство, если оно не существует

Никакого исключения не возникает

Если рассмотреть приведенный ниже фрагмент кода, то, как правило, следует ожидать появления исключения.

```
const rootFolder = new Folder();
rootFolder.ninjasDir.firstNinjaDir.ninjaFile = "yoshi.txt";
```

В данном фрагменте кода осуществляется доступ к свойству `firstNinjaDir` неопределенного свойства `ninjasDir` объекта `rootFolder`. Но если выполнить этот код из листинга 8.11, то окажется, что он вполне работоспособен, как показано на рис. 8.5.

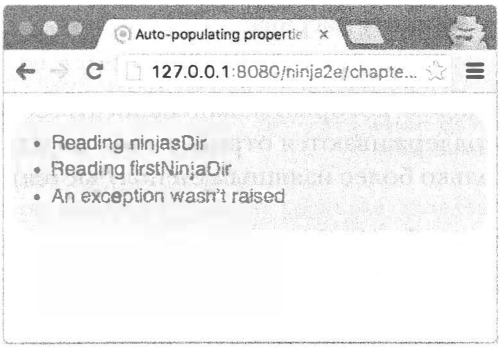


Рис. 8.5. Результат выполнения кода из листинга 8.11

Это происходит потому, что в данном примере кода применяется прокси-объект. Всякий раз, когда осуществляется доступ к свойству, в прокси-объекте активизируется метод перехвата `get()`. И если объект папки уже содержит запрашиваемое свойство, то возвращается его значение. А если запрашиваемое свойство отсутствует, то создается новая папка, которая присваивается данному свойству. Именно таким образом создаются два свойства: `ninjasDir` и `firstNinjaDir`. Если запрашивается значение неинициализированного свойства, то активизируется его создание. Наконец-то мы получили в свое распоряжение средство, позволяющее избежать неприятных исключений при доступе к несуществующим свойствам!

8.2.4. Применение прокси-объектов для реализации отрицательных индексов массивов

В повседневной практике программирования нередко приходится уделять немало внимания обработке массивов. Выясним, как с выгодой воспользоваться прокси-объектами, чтобы сделать немного более удобной работу с массивами.

Если у вас имеется опыт программирования на таких языках, как Python, Ruby или Perl, то вам, вероятно, приходилось пользоваться отрицательными индексами для доступа к элементам с конца массива, как показано в следующем фрагменте кода:

```
const ninjas = ["Yoshi", "Kuma", "Hattori"];

ninjas[0]; // "Yoshi"
ninjas[1]; // "Kuma"
ninjas[2]; // "Hattori"
```

Обычный доступ к элементам массива по положительным индексам

```
ninjas[-1]; // "Hattori"
ninjas[-2]; // "Kuma"
ninjas[-3]; // "Yoshi"
```

Отрицательные индексы позволяют получать доступ к элементам с конца массива, начиная с индекса -1, указывающего на последний элемент массива

А теперь сравните следующий код, который обычно приходится писать для доступа к последнему элементу массива:

```
ninjas[ninjas.length-1]
```

с приведенным выше кодом, который можно написать, если в избранном языке программирования поддерживаются отрицательные индексы массивов (обратите внимание, насколько более изящным он получается).

```
ninjas[-1]
```

К сожалению, в языке JavaScript отсутствует встроенная поддержка отрицательных индексов массивов, но ее можно сымитировать с помощью прокси-объектов. Чтобы исследовать такую возможность, рассмотрим несколько упрощенный вариант кода, написанного Синдре Сорхусом (Sindre Sorhus; `https://github.com/sindresorhus/negative-array`) и приведенного в листинге 8.12.

Листинг 8.12. Реализация отрицательных индексов массивов с помощью прокси-объектов

Если целевой объект не является массивом, сгенерировать исключение

Возвратить новый прокси-объект, которому передается массив, чтобы пользоваться им как целевым объектом

Преобразовать имя свойства в число с помощью унарной операции +

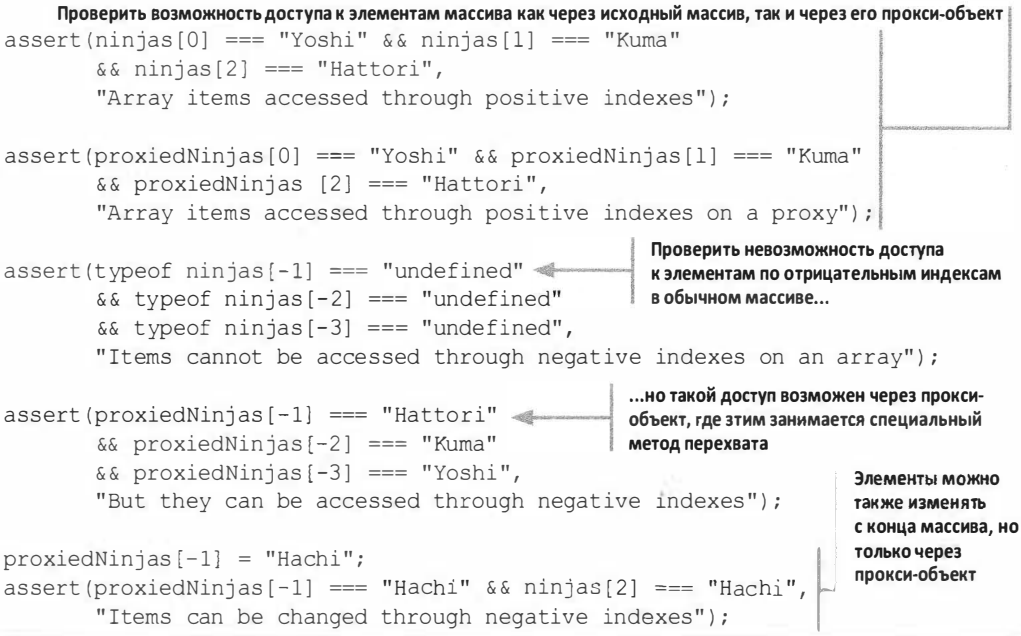
Если полученный индекс окажется отрицательным числом, прочитать массив с конца, а если индекс окажется положительным числом, то как обычно

Определить метод перехвата, активизируемый всякий раз, когда записывается индекс массива

Передать этот массив функции, создающей для него прокси-объект

Создать обычный массив

```
function createNegativeArrayProxy(array){
  if (!Array.isArray(array)) {
    throw new TypeError('Expected an array');
  }

  return new Proxy(array, {
    get: (target, index) => {
      index = +index;
      return target[index < 0 ? target.length + index : index];
    },
    set: (target, index, val) => {
      index = +index;
      return target[index < 0 ? target.length + index : index] = val;
    }
  });
}

const ninjas = ["Yoshi", "Kuma", "Hattori"];
const proxiedNinjas = createNegativeArrayProxy(ninjas);
```



Сначала в данном примере кода определяется функция, создающая прокси-объект для переданного ей массива. А поскольку этот прокси-объект не предназначен для других типов объектов, то генерируется исключение, если аргумент данной функции не является массивом:

```
if (!Array.isArray(array)) {
  throw new TypeError('Expected an array');
}
```

Затем в данном примере кода создается и возвращается новый прокси-объект с двумя методами перехвата. В частности, метод перехвата `get()` вызывается всякий раз, когда предпринимается попытка прочитать содержимое элемента массива. А метод перехвата `set()` активизируется всякий раз, когда в элемент массива осуществляется запись данных:

```
return new Proxy(array, {
  get: (target, index) => {
    index = +index;
    return target[index < 0 ? target.length + index : index];
  },
  set: (target, index, val) => {
    index = +index;
    return target[index < 0 ? target.length + index : index] = val;
  }
});
```

Тела обоих методов перехвата очень похожи. Сначала в них значение свойства преобразовывается в число с помощью унарной операции "плюс" (`index`

= +index). Затем доступ к элементам осуществляется с конца массива, если запрашиваемый индекс меньше нуля, или же обычным способом, если индекс массива больше нуля.

И, наконец, в рассматриваемом здесь примере кода выполняются различные тесты, чтобы убедиться, что в обычных массивах элементы могут быть доступны только по положительным индексам. А если используется прокси-объект, то получать доступ к элементам массива и видоизменять их содержимое можно и по отрицательным индексам.

Итак, показав, как пользоваться прокси-объектами для того, чтобы получить некоторые интересные возможности, в том числе автоматическое заполнение свойств объектов и доступ к элементам массива по отрицательным индексам, чего практически невозможно добиться без прокси-объектов, перейдем к рассмотрению вопросов производительности, в которых кроется самый существенный недостаток прокси-объектов.

8.2.5. Проблемы с производительностью при использовании прокси-объектов

Как пояснялось ранее, прокси-объект — это такой заменитель объекта, через который можно управлять доступом к другому объекту. В прокси-объекте можно определить методы перехвата, т.е. функции, которые будут выполняться всякий раз, когда над прокси-объектом выполняется определенная операция. И, как было показано ранее, этими методами перехвата можно воспользоваться для реализации таких полезных функциональных возможностей, как протоколирование, измерение производительности, автоматическое заполнение свойств объектов, реализация отрицательных индексов массивов и т.д. К сожалению, прокси-объекты не лишены недостатка. Вследствие того что операции должны проходить через прокси-объект, вносится определенный уровень косвенности, позволяющий реализовать все эти замечательные возможности. Но в то же время приходятся выполнять значительную часть дополнительной работы, что отрицательно сказывается на производительности.

Чтобы подробно исследовать вопросы производительности, рассмотрим в качестве примера создание отрицательных индексов массива из листинга 8.12 и сравним время выполнения при доступе к элементам массива обычным способом и через прокси-объект, как демонстрируется в примере кода из листинга 8.13.

Листинг 8.13. Выявление замедления в работе кода, вносимое прокси-объектом

```
function measure(items){
  const startTime = new Date().getTime();
  for(let i = 0; i < 500000; i++){
    items[0] === "Yoshi";
    items[1] === "Kuma";
    items[2] === "Hattori";
  }
```

Получить доступ к элементам массива в долго выполняющемся цикле

Получить текущее время перед выполнением длительной операции

```
  return new Date().getTime() - startTime;
}

const ninjas = ["Yoshi", "Kuma", "Hattori"];
const proxiedNinjas = createNegativeArrayProxy(ninjas);

console.log("Proxies are around",
            Math.round(measure(proxiedNinjas)/ measure(ninjas)),
            "times slower");
```

Измерить время, затрачиваемое на длительное выполнение кода

Сравнить текущее время при доступе к массиву стандартным способом и через прокси-объект

Создать обычный массив и его прокси-объект

Поскольку однократное выполнение операции в данном примере кода происходит слишком быстро, ее приходится выполнять многократно, чтобы определить точное значение времени задержки. Зачастую счет может идти на десятки тысяч и даже миллионы в зависимости от характера кода, производительность которого измеряется. Методом проб и ошибок было выбрано приемлемое значение **500000**.

Выполнение кода в данном примере происходит в промежутке между двумя отметками времени, получаемыми с помощью операции `new Date().getTime()`: одной — до начала выполнения целевого кода, а другой — после его выполнения. Разность этих отметок времени позволяет судить о том, как долго выполняется код. И, наконец, полученные результаты можно сравнить, вызвав метод `measure()` как для прокси-объекта, так и для обычного массива.

Результаты, полученные на нашем компьютере, свидетельствуют далеко не в пользу прокси-объектов. Так, в браузере Chrome прокси-объекты действуют почти в 50 раз медленнее, а в браузере Firefox — почти в 20 раз.

В настоящий момент рекомендуется очень аккуратно пользоваться прокси-объектами. И хотя они позволяют творчески подходить к управлению доступом к объектам, тем не менее, такое управление дается не бесплатно, а за счет потери производительности. В частности, прокси-объекты можно свободно применять в коде, некритичном к производительности, но делать нужно это очень аккуратно в коде, требующем продолжительного выполнения. И, как всегда, рекомендуется тщательно проверять производительность своего кода.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Доступ к объектам можно контролировать с помощью методов получения и установки, а также прокси-объектов.
- С помощью методов доступа (т.е. методов получения и установки) можно управлять доступом к свойствам объектов.
  - Метод доступа к свойствам может быть определен с помощью встроенного метода `Object.defineProperty()`, специального синтаксиса ключевых слов `get` и `set` как составных частей литералов объектов или классов, введенных в стандарт ES6.

  - Метод `get()` неявно вызывается всякий раз, когда предпринимается попытка прочитать, а метод `set()` — присвоить значение свойству объекта.
  - Методы получения могут служить для определения вычисляемых свойств, т.е. таких свойств, значения которых вычисляются по запросу, тогда как методы установки — для проверки достоверности вводимых данных и протоколирования выводимых результатов вычислений.
- Прокси-объекты, введенные в стандарт ES6 языка JavaScript, служат для управления другими объектами.
  - Прокси-объекты позволяют определить специальные действия, которые будут выполняться при взаимодействии с целевым объектом (например, в том случае, когда получается значение свойства или вызывается функция).
  - Все взаимодействия должны проходить через прокси-объект, у которого имеются методы перехвата, запускающиеся при выполнении конкретной операции.
- Пользуйтесь прокси-объектами для изящной реализации:
  - протоколирования;
  - измерения производительности;
  - проверки достоверности данных;
  - автоматического заполнения свойств объектов, чтобы избежать неприятных исключений из-за доступа к неинициализированным свойствам;
  - отрицательных индексов массивов.
- Прокси-объекты вносят задержку, поэтому пользоваться ими в коде, требующем продолжительного выполнения, следует очень аккуратно. Но в любом случае рекомендуется тщательно проверять производительность своего кода.

Упражнения

1. В каком из выражений в двух последних строках приведенного ниже фрагмента кода могут быть сгенерированы исключения при его выполнении и почему это может произойти?

```
const ninja = {
  get name() {
    return "Akiyama";
  }
}

ninja.name();
const name = ninja.name;
```

2. Какой механизм позволяет методу получения обратиться к закрытой объектной переменной в приведенном ниже фрагменте кода?

```
function Samurai() {
  const _weapon = "katana";
  Object.defineProperty(this, "weapon", {
    get: () => _weapon
  });
}
const samurai = new Samurai();
assert(samurai.weapon === "katana", "A katana wielding samurai");
```

3. Какое из приведенных ниже утверждений пройдет?

```
const daimyo = { name: "Matsu", clan: "Takasu"};
const proxy = new Proxy(daimyo, {
  get: (target, key) => {
    if(key === "clan"){
      return "Tokugawa";
    }
  }
});

assert(daimyo.clan === "Takasu", "Matsu of clan Takasu");
assert(proxy.clan === "Tokugawa", "Matsu of clan Tokugawa?");

proxy.clan = "Tokugawa";

assert(daimyo.clan === "Takasu", "Matsu of clan Takasu");
assert(proxy.clan === "Tokugawa", "Matsu of clan Tokugawa?");
```

4. Какое из приведенных ниже утверждений пройдет?

```
const daimyo = { name: "Matsu", clan: "Takasu", armySize: 10000};
const proxy = new Proxy(daimyo, {
  set: (target, key, value) => {
    if(key === "armySize") {
      const number = Number.parseInt(value);
      if(!Number.isNaN(number)){
        target[key] = number;
      }
    } else {
      target[key] = value;
    }
  },
});

assert(daimyo.armySize === 10000, "Matsu has 10 000 men at arms");
assert(proxy.armySize === 10000, "Matsu has 10 000 men at arms");

proxy.armySize = "large";
assert(daimyo.armySize === "large", "Matsu has a large army");

daimyo.armySize = "large";
assert(daimyo.armySize === "large", "Matsu has a large army");
```

9 Работа с коллекциями

В этой главе...

- Создание и видоизменение массивов
- Применение и повторное использование функций массивов
- Создание словарей с помощью отображений
- Создание коллекций уникальных объектов с помощью множеств

Итак, уделив достаточно времени обсуждению объектно-ориентированных особенностей в JavaScript, перейдем к рассмотрению тесно связанных с ними вопросов организации и обработки коллекций элементов. И начнем мы с массивов — самого распространенного типа коллекций в JavaScript. Мы рассмотрим ряд особенностей массивов, которые могут оказаться неожиданными для тех, у кого имеется опыт программирования на других языках. А продолжим мы с исследования набора встроенных методов, способных оказать помощь в написании более изящного кода обработки массивов.

Далее мы обсудим два новых типа коллекций, введенных в стандарт ES6: отображения и множества. С одной стороны, отображения позволяют создавать словари со взаимным соответствием ключей и значений, т.е. такие коллекции, которые чрезвычайно полезны для решения некоторых задач программирования. А с другой стороны, множества являются коллекциями уникальных элементов, где каждый элемент может встречаться только один раз. Начнем наше исследование с массивов — самого простого и распространенного типа коллекций.

Знаете ли вы?

Какие типичные препятствия таятся в применении объектов в качестве словарей или отображений?

Какие типы значений может иметь пара "ключ–значение" в отображении `Map`?

Должны ли элементы множества иметь одинаковый тип?

9.1. Массивы

Массивы относятся к самым распространенным типам данных. С их помощью можно организовать обработку элементов коллекций. Если у вас имеется опыт программирования на таком строго типизированном языке, как C, то вероятно, массивы вы рассматриваете как последовательные ячейки оперативной памяти, где хранятся однотипные элементы, а каждая ячейка оперативной памяти имеет фиксированный размер и соответствующий индекс, по которому они могут быть легко доступны.

Но, как и многие другие языковые средства, массивы имеют в JavaScript свою особенность: они являются объектами. И хотя эта особенность приводит к некоторым злополучным побочным эффектам, главным образом, с точки зрения производительности, тем не менее она дает и ряд преимуществ. Например, массивам, как и любым другим объектам, могут быть доступны методы, которые способны упростить задачу разработчикам приложений на JavaScript.

Сначала мы рассмотрим в этом разделе способы создания массивов. Затем выясним, каким образом элементы вводятся и удаляются в разных местах массива. И, наконец, исследуем встроенные методы, позволяющие сделать намного более изящным код обработки массивов.

9.1.1. Создание массивов

Имеются два основных способа создания новых массивов с помощью:

- встроенного конструктора объектов типа `Array`;
- литералов массивов `[]`.

Начнем с простого примера, в котором создаются массивы имен ниндзя и самураев (листинг 9.1).

Листинг 9.1. Создание массивов

Чтобы создать массив, можно воспользоваться литералом массива `[]` ...

... или встроенным конструктором объектов типа `Array`

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
const samurai = new Array("Oda", "Tomoe");
```

Свойство `length` сообщает размер массива

```
assert(ninjas.length === 3, "There are three ninjas");
assert(samurai.length === 2, "And only two samurai");
```

Доступ к элементам массива по индексу. Первый элемент массива доступен по индексу 0, а последний элемент — по индексу `array.length - 1`

```
assert(ninjas[0] === "Kuma", "Kuma is the first ninja");
assert(samurai[samurai.length-1] === "Tomoe",
       "Tomoe is the last samurai");

assert(ninjas[4] === undefined,
       "We get undefined if we try to access
        an out of bounds index");

ninjas[4] = "Ishi";
assert(ninjas.length === 5,
       "Arrays are automatically expanded");

ninjas.length = 2;
assert(ninjas.length === 2, "There are only two ninjas now");
assert(ninjas[0] === "Kuma" && ninjas[1] === "Hattori",
       "Kuma and Hattori");
assert(ninjas[2] === undefined, "But we've lost Yagyu");
```

Если обратиться к элементам за пределы массива, в итоге будет получено неопределенное значение (`undefined`)

Попытка осуществить запись за пределами массива приводит к его расширению

Попытка переопределить свойство `length` меньшим значением приводит к удалению лишних элементов

Вначале кода из листинга 9.1 создаются два массива. В частности, массив `ninjas` создается с помощью простого литерала массива следующим образом:

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
```

Этот массив сразу же заполняется тремя именами ниндзя: `Kuma`, `Hattori` и `Yagyu`. А массив `samurai` создается с помощью встроенного конструктора объектов типа `Array` таким образом:

```
const samurai = new Array("Oda", "Tomoe");
```

Литералы массивов в сравнении с конструктором объектов типа `Array`

Литералы массивов более предпочтительны для создания массивов по сравнению с конструктором объектов типа **`Array`**. И главная причина заключается в простоте записи: **`[]`** против **`new Array()`**, т.е. 2 символа оказываются вне конкуренции при сравнении с 11 символами. Кроме того, ничто не мешает переопределить встроенный конструктор объектов типа **`Array`** в силу высокой динамичности языка JavaScript. А это означает, что при вызове данного конструктора в операции **`new Array()`** совсем не обязательно должен создаваться массив. Поэтому рекомендуется, как правило, отдавать предпочтение литералам массивов.

Независимо от способа создания, каждый массив обладает свойством `length`, обозначающим размер массива. Так, если значение свойства `length` равно **3**, то массив `ninjas` в коде из листинга 9.1 содержит три имени ниндзя. Данное обстоятельство можно проверить в следующих утверждениях:

```
assert(ninjas.length === 3, "There are three ninjas");
assert(samurai.length === 2, "And only two samurai");
```

Как вам должно быть уже известно, элементы доступны в массиве по индексному обозначению, где первый элемент находится по индексу **0**, а последний элемент — по индексу **`array.length - 1`**. Но если попытаться обратиться по индексу (например, `ninjas[4]`) за пределы массива, который содержит лишь три имени ниндзя, то в ответ не будет получено жуткое сообщение `"Array index out of bounds"` (Индекс находится за пределами массива) об исключительной ситуации, как это обычно происходит в большинстве других языков программирования. Вместо этого возвращается неопределенное значение (`undefined`), свидетельствующее о том, что по указанному индексу ничего нет:

```
assert(ninjas[4] === undefined,
       "We get undefined if we try to access an out of bounds index");
```

Такое поведение следует из того обстоятельства, что массивы в языке JavaScript являются объектами. Неопределенное значение (`undefined`) возвращается при доступе по индексу к несуществующему элементу массива таким же образом, как и при доступе к несуществующему свойству объекта.

Но если попытаться осуществить запись за пределами массива, как в следующей строке кода:

```
ninjas[4] = "Ishi";
```

то массив будет расширен с целью приспособиться к новой ситуации. В таком случае в данном массиве, по существу, образуется брешь, а элемент по индексу **3** не определен (`undefined`), как показано на рис. 9.1. При этом изменяется также значение в свойстве `length`, которое теперь равно **5**, несмотря на то, что один элемент данного массива оказывается неопределенным.

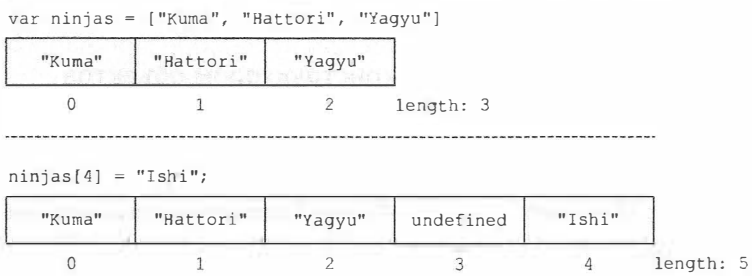


Рис. 9.1. Запись по индексу за пределами массива

В отличие от большинства других языков, массивам в JavaScript присуща еще одна особенность, связанная со свойством `length`. В частности, ничто не мешает изменить вручную значение этого свойства. Если установить в свойстве `length` значение, большее его текущего значения, то массив расширится неопределенными (`undefined`) элементами, а если установить меньшее значение, массив сократится, как показано ниже.

```
ninjas.length = 2;
```

А теперь, когда стали понятны основы создания массивов, перейдем к рассмотрению некоторых из наиболее употребительных методов их обработки.

9.1.2. Добавление и удаление элементов с обоих концов массива

Рассмотрим сначала перечисленные ниже простые методы, с помощью которых можно добавлять элементы в массив и удалять их из него.

- Метод `push()` — добавляет элемент в конец массива.
- Метод `unshift()` — добавляет элемент в начало массива.
- Метод `pop()` — удаляет элемент из конца массива.
- Метод `shift()` — удаляет элемент из начала массива.

Вам, вероятно, уже приходилось пользоваться этими методами, но на всякий случай, ради целостности изложения материала, рассмотрим их применение на примере кода из листинга 9.2.

Листинг 9.2. Добавление и удаление элементов массива

```
const ninjas = [];
assert(ninjas.length === 0, "An array starts empty");
ninjas.push("Kuma");
assert(ninjas[0] === "Kuma",
       "Kuma is the first item in the array");
assert(ninjas.length === 1, "We have one item in the array");

ninjas.push("Hattori");
assert(ninjas[0] === "Kuma",
       "Kuma is still first");
assert(ninjas[1] === "Hattori",
       "Hattori is added to the end of the array");
assert(ninjas.length === 2,
       "We have two items in the array!");

ninjas.unshift("Yagyu");
assert(ninjas[0] === "Yagyu",
       "Now Yagyu is the first item");
assert(ninjas[1] === "Kuma",
       "Kuma moved to the second place");
assert(ninjas[2] === "Hattori",
       "And Hattori to the third place");
assert(ninjas.length === 3,
       "We have three items in the array!");

const lastNinja = ninjas.pop();
assert(lastNinja === "Hattori",
       "We've removed Hattori from the end of the array");
assert(ninjas[0] === "Yagyu",
       "Now Yagyu is still the first item");
assert(ninjas[1] === "Kuma,
       "Kuma is still in second place");
assert(ninjas.length === 2,
       "Now there are two items in the array");
```

Создать новый пустой массив

Добавить новый элемент в конец массива

Добавить еще один элемент в конец массива

Добавить элемент в начало массива с помощью встроенного метода `unshift()`. Соответственно корректируется положение остальных элементов массива

Удалить последний элемент из массива

```
const firstNinja = ninjas.shift();
assert(firstNinja === "Yagyu",
       "We've removed Yagyu from the beginning of the array");
assert(ninjas[0] === "Kuma",
       "Kuma has shifted to the first place");
assert(ninjas.length === 1,
       "There's only one ninja in the array");
```

Удалить первый элемент из массива. Соответственно корректируется положение остальных элементов массива

Сначала в данном примере кода создается новый пустой массив `ninjas` следующим образом:

```
ninjas = [] // ninjas: []
```

Для каждого массива можно вызвать встроенный метод `push()`, чтобы добавить новый элемент в конец данного массива, изменив по ходу дела его длину:

```
ninjas.push("Kuma"); // ninjas: ["Kuma"];
ninjas.push("Hattori"); // ninjas: ["Kuma", "Hattori"];
```

Чтобы добавить новые элементы в начало массива, достаточно вызвать встроенный метод `unshift()` следующим образом:

```
ninjas.unshift("Yagyu");// ninjas: ["Yagyu", "Kuma", "Hattori"];
```

Обратите внимание на то, что при этом корректируется положение элементов, уже существующих в массиве. Например, до вызова метода `unshift()` элемент со строковым значением `"Kuma"` находился в массиве `ninjas` по индексу **0**, а после вызова данного метода этот элемент уже находится по индексу **1**.

Кроме того, элементы можно удалить как из конца, так и из начала массива. В результате вызова встроенного метода `pop()` удаляется элемент из конца массива и по ходу дела сокращается длина массива:

```
var lastNinja = ninjas.pop(); // ninjas: ["Yagyu", "Kuma"]
                              // lastNinja: "Hattori"
```

А вызвав встроенный метод `shift()`, можно удалить элемент из начала массива следующим образом:

```
var firstNinja = ninjas.shift(); // ninjas: ["Kuma"]
                                 // firstNinja: "Yagyu"
```

На рис. 9.2 наглядно показано, каким образом методы `push()`, `pop()`, `shift()` и `unshift()` видоизменяют массив.

Сравнение методов `pop()` и `push()` с методами `shift()` и `unshift()` с точки зрения производительности

С одной стороны, методы **`pop()`** и **`push()`** оказывают влияние только на последний элемент массива, соответственно удаляя и добавляя элемент в его конце. А с другой стороны, методы **`shift()`** и **`unshift()`** оказывают соответствующее влияние на первый элемент массива, в результате чего корректируются индексы любых последующих элементов массива. Именно по этой причине

методы `pop()` и `push()` действуют намного быстрее, чем методы `shift()` и `unshift()`, а следовательно, предпочтение следует отдавать первым, если нет веских оснований поступить иначе.

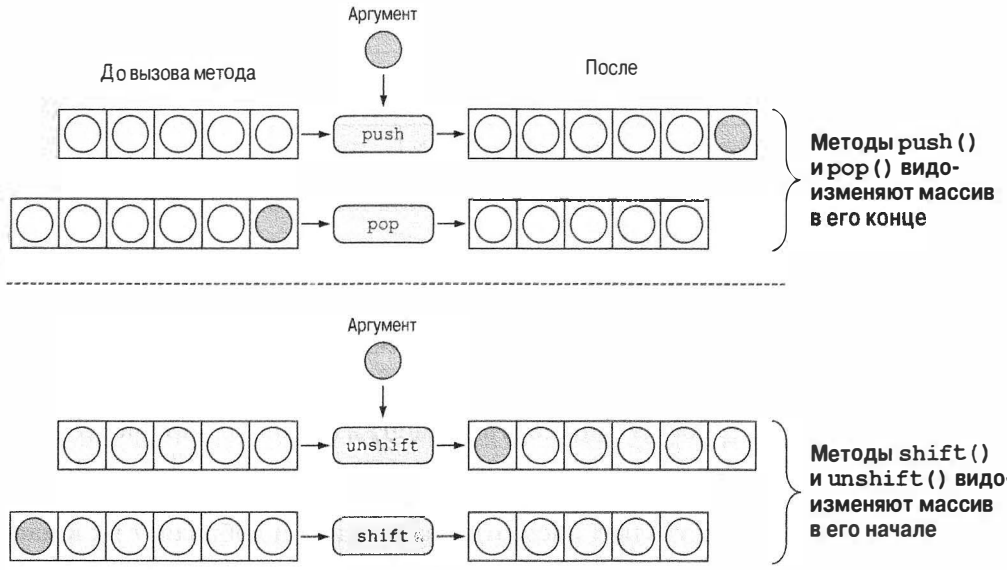


Рис. 9.2. Методы `push()` и `pop()` видоизменяют массив в его конце, тогда как методы `shift()` и `unshift()` — в начале

9.1.3. Добавление и удаление элементов в любом месте массива

В предыдущем примере кода элементы удалялись из начала и из конца массива. Но такие операции слишком ограничены в целом, поскольку должна быть возможность удалять элементы в любом месте массива. Один из самых простых методов сделать это демонстрируется в примере кода из листинга 9.3.

Листинг 9.3. Простейший способ удаления элемента из массива

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

delete ninjas[1];

assert(ninjas.length === 4,
       "Length still reports that there are 4 items");

assert(ninjas[0] === "Yagyu", "First item is Yagyu");
assert(ninjas[1] === undefined, "We've simply created a hole");
assert(ninjas[2] === "Hattori", "Hattori is still the third item");
assert(ninjas[3] === "Fuma", "And Fuma is the last item");
```

Удалить элемент по команде `delete`

Несмотря на то что элемент удален, в массиве по-прежнему остаются 4 элемента. В нем образовалась лишь брешь

Такой способ удаления элемента из массива оказывается недействительным. Ведь при этом в массиве, по существу, образуется брешь. И хотя из массива `ninjas` в приведенном выше примере кода формально удален один элемент, в нем по-прежнему остаются четыре элемента, причем тот элемент, который должен быть удален, на самом деле остается на своем месте, только в неопределенном (`undefined`) состоянии (рис. 9.3).

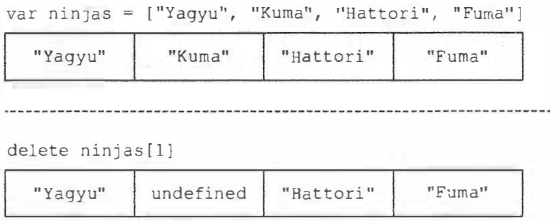


Рис. 9.3. В результате удаления элемента в массиве образуется брешь

Аналогично, если требуется добавить элемент в произвольное место массива, то непонятно, откуда следует начинать эту операцию? Для разрешения подобных затруднений в JavaScript имеется метод `splice()`, доступный всем массивам, который удаляет элементы из массива и добавляет их в массив, начиная с указанного индекса. В качестве примера рассмотрим код из листинга 9.4.

Листинг 9.4. Удаление и добавление элементов в произвольных местах массива

```
const ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];

var removedItems = ninjas.splice(1, 1);

assert(removedItems.length === 1, "One item was removed");
assert(removedItems[0] === "Kuma");

assert(ninjas.length === 3,
       "There are now three items in the array");
assert(ninjas[0] === "Yagyu",
       "The first item is still Yagyu");
assert(ninjas[1] === "Hattori",
       "Hattori is now in the second place");
assert(ninjas[2] === "Fuma",
       "And Fuma is in the third place");

removedItems = ninjas.splice(1, 2, "Mochizuki", "Yoshi", "Momochi");
assert(removedItems.length === 2, "Now, we've removed two items");
assert(removedItems[0] === "Hattori", "Hattori was removed");
assert(removedItems[1] === "Fuma", "Fuma was removed");
assert(ninjas.length === 4, "We've inserted some new items");
```

Создать новый массив с четырьмя элементами

Вызвать встроенный метод `splice()`, чтобы удалить один элемент, начиная с места по указанному индексу

Метод `splice()` возвращает массив удаленных элементов (в данном случае — один удаленный элемент)

Массив `ninja` больше не содержит имя `Kuma`, а последующие элементы массива автоматически сдвинуты

```
assert(ninjas[0] === "Yagyu", "Yagyu is still here");
assert(ninjas[1] === "Mochizuki", "Mochizuki also");
assert(ninjas[2] === "Yoshi", "Yoshi also");
assert(ninjas[3] === "Momochi", "and Momochi");
```

Элемент можно добавить в любое место массива, указав дополнительные аргументы при вызове метода `splice()`

Сначала в данном примере кода создается новый массив, состоящий из четырех элементов:

```
var ninjas = ["Yagyu", "Kuma", "Hattori", "Fuma"];
```

Затем вызывается встроенный метод `splice()`:

```
var removedItems = ninjas.splice(1,1);
  // ninjas: ["Yagyu","Hattori", "Fuma"];
  // removedItems: ["Kuma"]
```

В данном случае методу `splice()` передается два аргумента: индекс места, с которого начинается сращивание элементов массива, а также количество удаляемых элементов (если опустить этот аргумент, будут удалены все элементы до конца массива). Таким образом, из массива удаляется элемент по индексу **1**, а все последующие элементы соответственно сдвигаются.

Кроме того, метод `splice()` возвращает массив удаленных элементов. В данном случае это массив, состоящий из единственного элемента `"Kuma"`. С помощью метода `splice()` можно также добавлять элементы в произвольные места массива. Рассмотрим в качестве примера приведенный ниже фрагмент кода. Начиная с места по индексу **1**, в этом примере кода сначала удаляются два элемента, а затем добавляются три элемента массива: `"Mochizuki"`, `"Yoshi"` и `"Momochi"`.

```
removedItems = ninjas.splice(1, 2, "Mochizuki", "Yoshi", "Momochi");
  // ninjas: ["Yagyu", "Mochizuki", "Yoshi", "Momochi"]
  // removedItems: ["Hattori", "Fuma"]
```

Итак, напомнив, каким образом действуют массивы, продолжим исследовать некоторые наиболее употребительные операции, выполняемые над массивами. Это должно оказать вам помощь в написании более изящного кода обработки массивов.

9.1.4. Наиболее употребительные операции над массивами

В этом разделе исследуются некоторые из наиболее употребительных операций над массивами, в том числе следующие.

- **Обход** (или **перебор**) элементов массива.
- **Отображение** существующих элементов массива для создания на их основе нового массива.
- **Тестирование** элементов массива, чтобы проверить, удовлетворяют ли они определенным условиям.

- **Поиск** конкретных элементов массива.
- **Агрегирование** массивов и вычисление единственного значения на основе элементов массива (например, расчет суммы значений в массиве).

Начнем с самой простой операции — перебора элементов массива.

Перебор элементов массива

Перебор элементов массива относится к числу наиболее употребительных операций над массивами. Если обратиться к азбучным истинам вычислительной техники, то можно вспомнить, что перебор элементов чаще всего производится следующим образом:

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

for(let i = 0; i < ninjas.length; i++){
  assert(ninjas[i] !== null, ninjas[i]);
}
```

Вывести имя ниндзя, находящееся в каждом элементе массива

На первый взгляд, код из приведенного выше примера выглядит довольно просто. В нем организуется цикл `for` для проверки содержимого каждого элемента в массиве, а результаты выполнения данного кода показаны на рис. 9.4.

Вам, вероятно, приходилось не раз писать подобный код, даже не задумываясь над ним. Но на всякий случай проанализируем цикл `for` более подробно.

Чтобы обойти массив, придется установить переменную счетчика `i`, указать число, до которого требуется произвести подсчет (`ninjas.length`), а также определить порядок изменения счетчика (`i++`). Таким образом, для выполнения столь распространенной операции приходится производить немало служебных действий, которые могут стать источником мелких, но неприятных программных ошибок. И все это, среди прочего, затрудняет чтение исходного кода, поскольку читающим его приходится анализировать каждую часть цикла `for`, чтобы убедиться, что при переборе массива не пропущен ни один из его элементов.

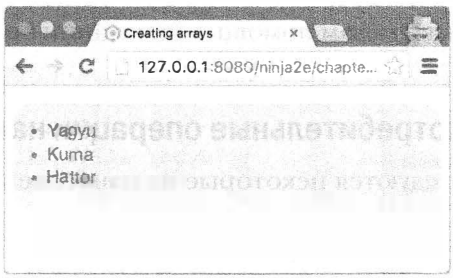


Рис. 9.4. Выводимые результаты проверки содержимого каждого элемента в массиве **`for`**

Чтобы упростить дело, все массивы в JavaScript снабжаются встроенным методом `forEach()`, которым можно воспользоваться в подобных случаях. В качестве примера рассмотрим код из листинга 9.5.

Листинг 9.5. Применение метода `forEach()`

```
const ninjas = ["Yagyu", "Kuma", "Hattori"];

ninjas.forEach(ninja => {
  assert(ninja !== null, ninja);
});
```

Воспользоваться встроенным методом `forEach()` для обхода массива

При обходе каждого элемента массива *немедленно* делается обратный вызов (в данном случае — стрелочной функции). А самое главное, что теперь не нужно беспокоиться о задании начального индекса, конечного условия или конкретного характера приращения при обходе массива, поскольку все это бремя берет на себя интерпретатор JavaScript. Обратите внимание, насколько проще понять рассматриваемый здесь код и в насколько меньшей степени он чреват программными ошибками.

А теперь рассмотрим более сложный вопрос отображения одних массивов на другие.

Отображение массивов

Допустим, имеется массив `ninjas` объектов, представляющих ниндзя, у каждого из которых имеется свое имя и излюбленный вид оружия. Из массива `ninjas` требуется извлечь массив видов оружия. Зная о наличии встроенного метода `forEach()`, для решения этой задачи можно написать код, аналогичный приведенному в листинге 9.6.

Листинг 9.6. Простейший способ извлечения массива видов оружия

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi", weapon: "katana"},
  {name: "Kuma", weapon: "wakizashi"}
];

const weapons = [];
ninjas.forEach(ninja => {
  weapons.push(ninja.weapon);
});
```

Создать новый массив и вызвать метод `forEach()`, чтобы извлечь виды оружия отдельных ниндзя, перебирая в цикле элементы исходного массива

```
assert(weapons[0] === "shuriken"
       && weapons[1] === "katana"
       && weapons[2] === "wakizashi"
       && weapons.length === 3,
       "The new array contains all weapons");
```

В данном примере кода не все так уж и плохо. Сначала в нем создается пустой массив, а затем вызывается метод `forEach()` для циклического обхода массива `ninjas`. При этом для каждого объекта `ninja` в массив `weapons` добавляется текущий вид оружия.

Нетрудно представить, что создание новых массивов из элементов существующего массива является настолько распространенной операцией, что она получила специальное название — *отображение* массива. Принцип действия данной операции состоит в том, чтобы отобразить каждый элемент из одного массива на соответствующий элемент нового массива, как демонстрируется в примере кода из листинга 9.7.

Листинг 9.7. Отображение массива

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi", weapon: "katana"},
  {name: "Kuma", weapon: "wakizashi"}
];

const weapons = ninjas.map(ninja => ninja.weapon);

assert(weapons[0] === "shuriken"
       && weapons[1] === "katana"
       && weapons[2] === "wakizashi"
       && weapons.length == 3,
       "The new array contains all weapons");
```

Встроенному методу `map()` передается функция, которая будет вызываться для каждого элемента массива

Встроенный метод `map()` сначала создает совершенно новый массив, а затем перебирает элементы исходного массива. Для каждого элемента исходного массива метод `map()` добавляет ровно один элемент во вновь созданный массив, исходя из результата работы функции обратного вызова, переданной методу `map()`. Внутренний механизм действия метода `map()` наглядно показан на рис. 9.5.

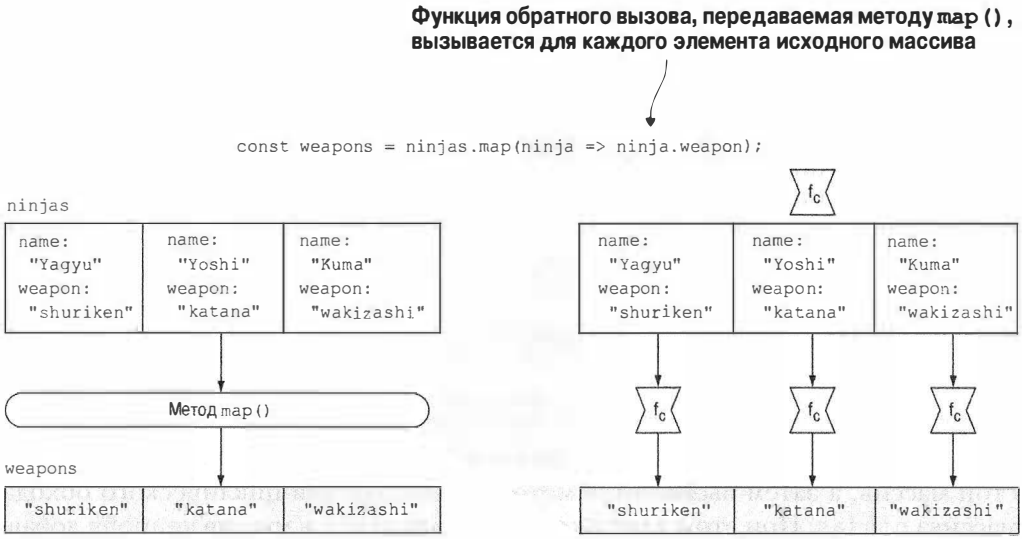


Рис. 9.5. Метод **`map()`** вызывает передаваемую ему функцию обратного вызова (**`fc`**) для каждого элемента массива и создает новый массив из значений, возвращаемых этой функцией

А теперь, зная, каким образом одни массивы отображаются на другие, выясним, как организовать проверку элементов массива по определенным условиям.

Проверка элементов массива

При обработке коллекций элементов нередко возникают такие ситуации, когда требуется выяснить, удовлетворяют ли *все* или хотя бы *некоторые* элементы массива определенным условиям. Для как можно более эффективного написания такого кода все массивы в JavaScript снабжаются встроенными методами `every()` и `some()`, как показано в примере кода из листинга 9.8.

Листинг 9.8. Проверка элементов массива с помощью методов `every()` и `some()`

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi" },
  {name: "Kuma", weapon: "wakizashi"}
];

const allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
const allNinjasAreArmed = ninjas.every(ninja => "weapon" in ninja);

assert(allNinjasAreNamed, "Every ninja has a name");
assert(!allNinjasAreArmed, "But not every ninja is armed");

const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
assert(someNinjasAreArmed, "But some ninjas are armed");
```

Встроенному методу `every()` передается функция обратного вызова, которая будет вызываться для каждого элемента массива. Он возвращает логическое значение `true`, если функция обратного вызова возвращает логическое значение `true` для *всех* элементов массива, а иначе — логическое значение `false`

Встроенному методу `some()` также передается функция обратного вызова. Он возвращает логическое значение `true`, если функция обратного вызова возвращает логическое значение `true` хотя бы для *одного* элемента массива, а иначе — логическое значение `false`

В данном примере кода имеется массив объектов, представляющих ниндзя, но не вполне известны их имена и вооружение. Чтобы в корне разрешить данное затруднение, сначала вызывается метод `every()`:

```
const allNinjasAreNamed = ninjas.every(ninja => "name" in ninja);
```

Методу `every()` передается функция обратного вызова, которая будет вызвана для каждого элемента массива. В ней проверяется, содержится ли имя каждого ниндзя в исходном массиве. Метод `every()` возвращает логическое значение `true` лишь в том случае, если переданная ему функция обратного вызова возвратит логическое значение `true` для *каждого* проверяемого элемента в массиве. Принцип действия метода `every()` показан на рис. 9.6.

Но иногда требуется выяснить, удовлетворяют ли заданному условию только *некоторые* элементы массива. В подобных случаях встроенный метод `some()` можно вызвать следующим образом:

```
const someNinjasAreArmed = ninjas.some(ninja => "weapon" in ninja);
```

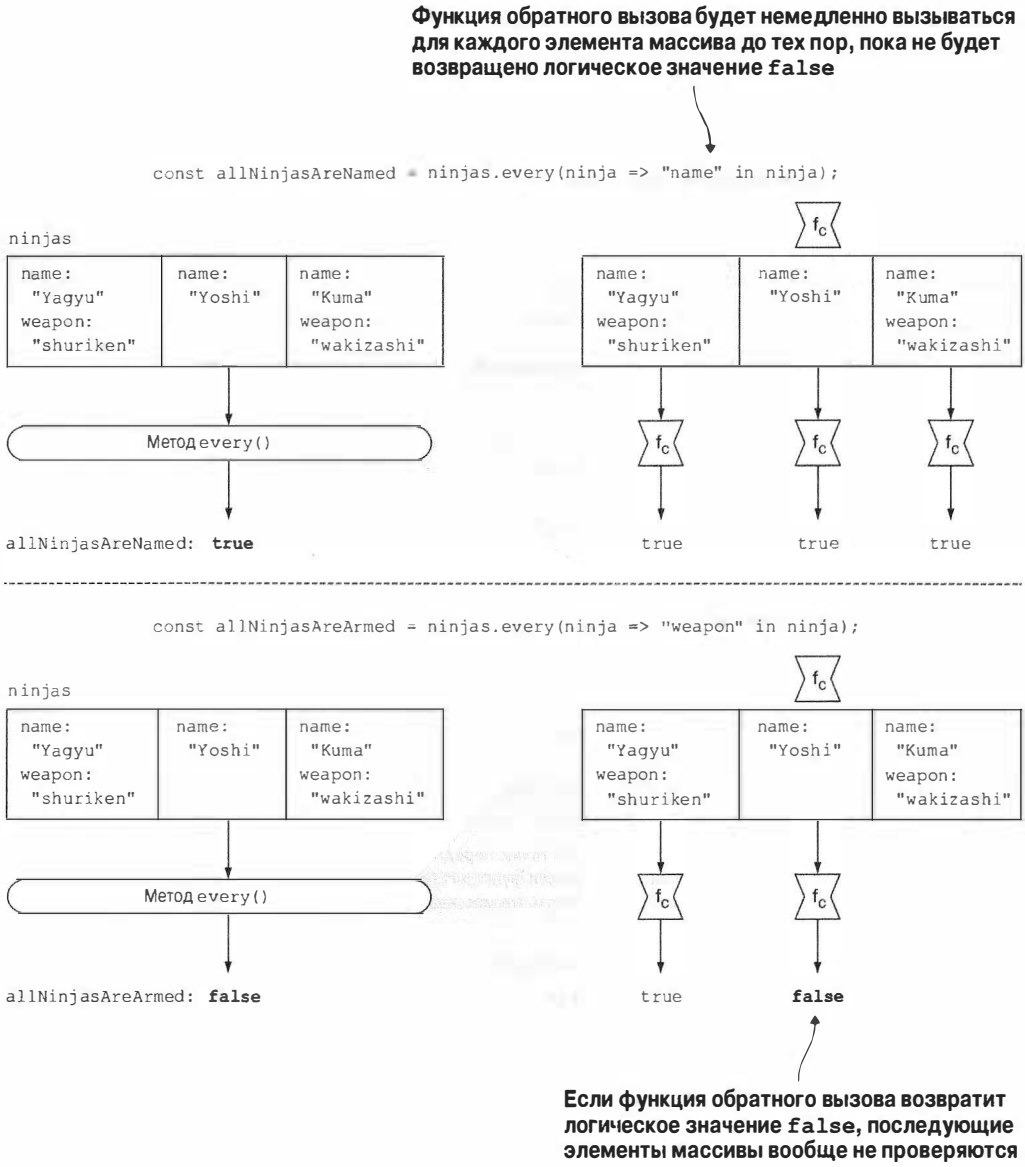


Рис. 9.6. В методе **every()** проверяется, удовлетворяют ли все элементы массива определенному условию, заданному в функции обратного вызова (**fc**)

Начиная с первого элемента массива, в методе `some()` делается обратный вызов для каждого элемента массива до тех пор, пока функция обратного вызова не возвратит логическое значение `true`. Если искомый элемент будет найден, то возвратится логическое значение `true`, а иначе — логическое значение `false`.

Принцип действия метода `some()` показан на рис. 9.7. Поиск в массиве производится с целью выяснить, удовлетворяют ли заданному условию некоторые или все элементы массива. Рассмотрим далее, как организовать поиск в массиве конкретного элемента.

Рис. 9.7. В методе **some()** проверяется, удовлетворяет ли хотя бы один элемент массива условию, заданному в функции обратного вызова (**fc**), переданной данному методу

Поиск отдельных элементов в массиве

Еще одной весьма распространенной операцией, к которой рано или поздно приходится обращаться, является поиск отдельных элементов в массиве. И эту задачу существенно упрощает встроенный метод `find()`, которым снабжаются все массивы в JavaScript. В качестве примера рассмотрим код из листинга 9.9.

На заметку

Встроенный метод **find()** относится к стандарту ES6. Его текущую поддержку в браузерах см. по адресу **http://kangax.github.io/compat-table/es6/#test-Array.prototype_methods_Array.prototype.find**.

Листинг 9.9. Поиск отдельных элементов в массиве

```
const ninjas = [
  {name: "Yagyu", weapon: "shuriken"},
  {name: "Yoshi" },
  {name: "Kuma", weapon: "wakizashi"}
];
```

```
const ninjaWithWakizashi = ninjas.find(ninja => {
  return ninja.weapon === "wakizashi";
});

assert(ninjaWithWakizashi.name === "Kuma"
       && ninjaWithWakizashi.weapon === "wakizashi",
       "Kuma is wielding a wakizashi");

const ninjaWithKatana = ninjas.find(nina => {
  return ninja.weapon === "katana";
});

assert(ninjaWithKatana === undefined,
       "We couldn't find a ninja that wields a katana");

const armedNinjas = ninjas.filter(ninja => "weapon" in ninja);

assert(armedNinjas.length === 2, "There are two armed ninjas:");
assert(armedNinjas[0].name === "Yagyu"
       && armedNinjas[1].name === "Kuma", "Yagyu and Kuma");
```

Вызвать метод `find()`, чтобы найти в массиве только один элемент, удовлетворяющий определенному условию, заданному в функции обратного вызова

Метод `find()` возвращает неопределенное значение (`undefined`), если элемент не удается найти в массиве

Вызвать метод `filter()`, чтобы найти в массиве несколько элементов, удовлетворяющих определенному условию

Найти в массиве отдельный элемент, удовлетворяющий определенному условию, совсем не трудно. Для этого достаточно вызвать встроенный метод `find()`, передав ему функцию обратного вызова, которая вызывается для каждого элемента массива до тех пор, пока искомый элемент не будет найден, на что указывает логическое значение `true`, возвращаемое данной функцией. Например, в выражении

```
ninjas.find(ninja => ninja.weapon === "wakizashi");
```

находится `Kuma` — первый же ниндзя из массива `ninjas`, владеющий коротким мечом вакидзаси.

Если при обходе всего массива искомый элемент так и не обнаружится, то конечным результатом поиска будет неопределенное значение (`undefined`). Например, в результате выполнения следующей строки кода:

```
ninjaWithKatana = ninjas.find(ninja => ninja.weapon === "katana");
```

возвратится значение `undefined`, поскольку в массиве `ninjas` отсутствует ниндзя, владеющий длинным мечом катана. Внутренний механизм действия метода `find()` показан на рис. 9.8.

Если же в массиве требуется найти несколько элементов, удовлетворяющих определенному критерию, достаточно вызвать метод `filter()`, создающий *новый* массив, который содержит все элементы, удовлетворяющие данному критерию. Например, в выражении

```
const armedNinjas = ninjas.filter(ninja => "weapon" in ninja);
```

создается новый массив `armedNinjas`, содержащий только те ниндзя, которые владеют оружием. В нем отсутствует бедный ниндзя `Yoshi`, у которого нет оружия. Принцип действия метода `filter()` показан на рис. 9.9.

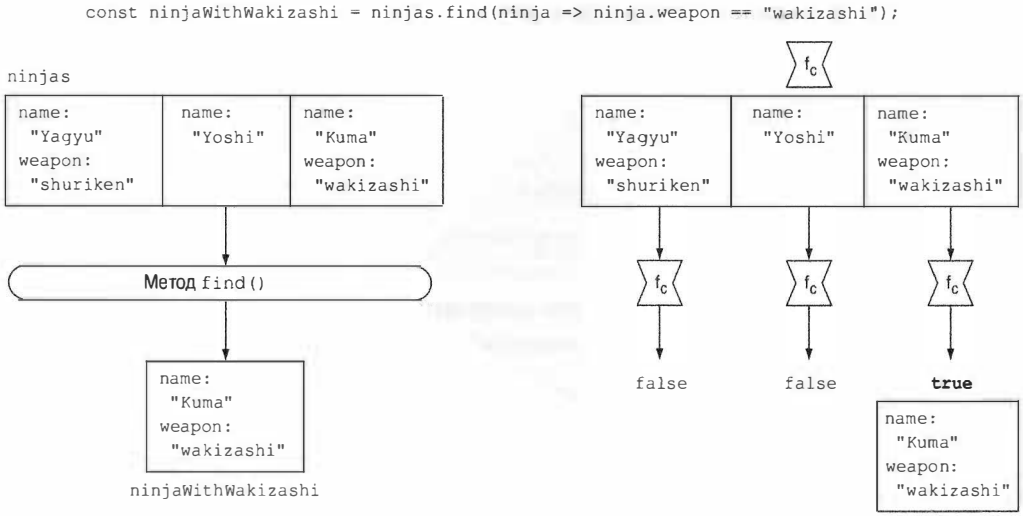


Рис. 9.8. Метод **find()** находит в массиве один элемент, т.е. первый элемент, для которого функция обратного вызова (**fc**), передаваемая методу **find()**, возвращает логическое значение **true**

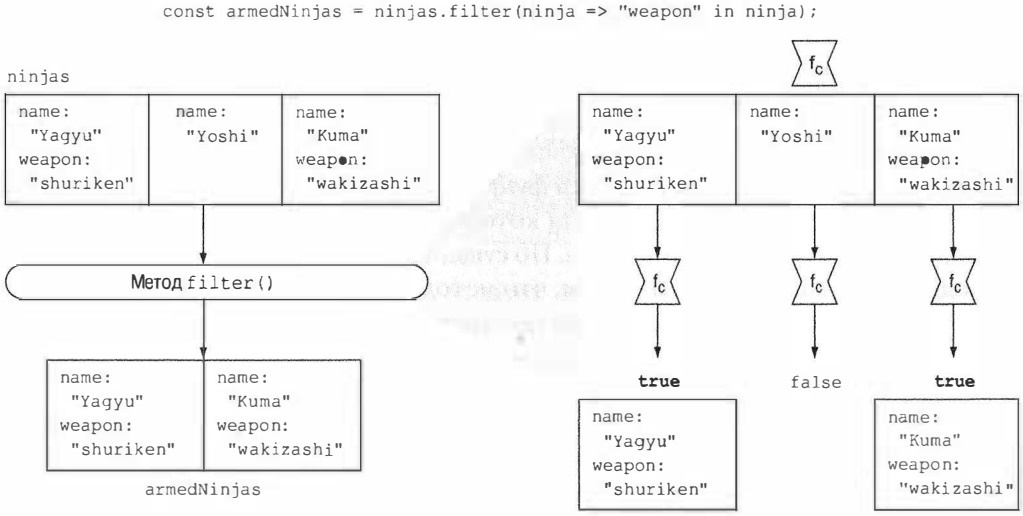


Рис. 9.9. В методе **filter()** создается новый массив, содержащий все элементы, для которых функция обратного вызова (**fc**) возвращает логическое значение **true**

В рассмотренном выше примере было показано, как находить отдельные элементы в массиве, но зачастую возникает также потребность определить индекс элемента массива. Рассмотрим эту задачу на примере кода из листинга 9.10.

Листинг 9.10. Определение индексов в массиве

```
const ninjas = ["Yagyu", "Yoshi", "Kuma", "Yoshi"];

assert(ninjas.indexOf("Yoshi") === 1, "Yoshi is at index 1");
assert(ninjas.lastIndexOf("Yoshi") === 3, "and at index 3");

const yoshiIndex = ninjas.findIndex(ninja => ninja === "Yoshi");

assert(yoshiIndex === 1, "Yoshi is still at index 1");
```

Чтобы узнать индекс отдельного элемента в массиве, достаточно вызвать встроенный метод `indexOf()`, передав ему элемент, индекс которого требуется определить:

```
ninjas.indexOf("Yoshi")
```

В тех случаях, когда отдельный элемент может встречаться в массиве несколько раз, как это происходит с элементом `"Yoshi"` массива `ninjas`, возникает также интерес выяснить последний индекс, по которому находится элемент `"Yoshi"`. Для этого существует метод `lastIndexOf()`, который можно вызвать следующим образом:

```
ninjas.lastIndexOf("Yoshi")
```

И, наконец, в самом общем случае, когда отсутствует ссылка на конкретный элемент массива, индекс которого требуется определить, можно вызвать метод `findIndex()` следующим образом:

```
const yoshiIndex = ninjas.findIndex(ninja => ninja === "Yoshi");
```

Методу `findIndex()` передается функция обратного вызова, а он возвращает индекс первого же элемента, для которого функция обратного вызова возвращает логическое значение `true`. По существу, этот метод действует подобно методу `find()`, а отличается он тем, что метод `find()` возвращает конкретный элемент, тогда как метод `findIndex()` — индекс этого элемента.

Сортировка массивов

К числу самых распространенных операций с массивами относится *сортировка* — систематическое расположение элементов массива в определенном порядке. К сожалению, реализация алгоритмов сортировки надлежащим образом является непростой задачей программирования. Ведь для решения подобной задачи необходимо выбрать наилучший алгоритм сортировки, реализовать его и приспособить к конкретным потребностям, действуя, как всегда, аккуратно, чтобы не внести едва уловимые программные ошибки. Чтобы снять бремя решения этой задачи с разработчиков, как было показано в главе 3, все массивы в JavaScript снабжены встроенным методом `sort()`, который вызывается следующим образом:

```
array.sort((a, b) => a - b);
```

Алгоритм сортировки реализован в интерпретаторе JavaScript. А нам остается лишь предоставить функцию обратного вызова, извещающую алгоритм сортировки о взаимосвязи, существующей между двумя элементами массива. Ниже перечислены возможные результаты вызова метода `sort()`.

- Если функция обратного вызова возвращает значение меньше нуля, элемент `a` должен предшествовать элементу `b`.
- Если функция обратного вызова возвращает нулевое значение, элементы `a` и `b` занимают одинаковое положение.
- Если функция обратного вызова возвращает значение больше нуля, элемент `a` должен следовать после элемента `b`.

Решения, принимаемые алгоритмом сортировки в зависимости от значения, возвращаемого функцией обратного вызова, приведены на рис. 9.10.

	... a ... b	... b ... a
`returnValue < 0` (элемент `a` должен предшествовать элементу `b`)	Оставить как есть	Элемент `a` следует расположить перед элементом `b`
`returnValue == 0` (элементы `a` и `b` занимают одинаковое положение)	Оставить как есть	Оставить как есть
`returnValue > 0` (элемент `b` должен предшествовать элементу `a`)	Элемент `b` следует расположить перед элементом `a`	Оставить как есть

Рис. 9.10. Если функция обратного вызова возвращает значение меньше нуля, первый сравниваемый элемент должен предшествовать второму. Если же данная функция возвращает нулевое значение, то оба элемента должны быть оставлены на месте. А если функция обратного вызова возвращает значение больше нуля, то первый сравниваемый элемент должен следовать после второго

Вот, собственно, и все, что следует знать об алгоритме сортировки. Конкретная сортировка выполняется внутри интерпретатора JavaScript и не требует от нас переставлять элементы массива вручную. В качестве простого примера рассмотрим код из листинга 9.11.

Листинг 9.11. Сортировка массива

```
const ninjas = [{name: "Yoshi"}, {name: "Yagyu"}, {name: "Kuma"}];

ninjas.sort(function(ninja1, ninja2){
  if(ninja1.name < ninja2.name) { return -1; }
  if(ninja1.name > ninja2.name) { return 1; }

  return 0;
});

assert(ninjas[0].name === "Kuma", "Kuma is first");
assert(ninjas[1].name === "Yagyu", "Yagyu is second");
assert(ninjas[2].name === "Yoshi", "Yoshi is third");
```

Передать функцию обратного вызова встроенному методу `sort()`, чтобы указать порядок сортировки

В коде из листинга 9.11 создается массив `ninjas` объектов, представляющих ниндзя, у каждого из которых имеется свое имя. Этот массив необходимо отсортировать в лексикографическом (или алфавитном) порядке по именам ниндзя. И с этой целью, естественно, вызывается метод `sort()`:

```
ninjas.sort(function(ninja1, ninja2){
  if(ninja1.name < ninja2.name) { return -1; }
  if(ninja1.name > ninja2.name) { return 1; }

  return 0;
});
```

Методу `sort()` достаточно передать функцию обратного вызова, применяемую для сравнения двух элементов массива. А поскольку требуется произвести лексикографическое сравнение, то в теле данной функции утверждается, что если имя первого ниндзя (объект `ninja1`) "меньше", чем имя второго ниндзя (объект `ninja2`), то функция обратного вызова возвращает значение **`-1`**, которое означает, что объект `ninja1` должен предшествовать объекту `ninja2` в окончательно отсортированном порядке. Если же имя первого ниндзя "больше", чем имя второго ниндзя, то функция обратного вызова возвращает значение **`1`**, которое означает, что объект `ninja1` должен следовать после объекта `ninja2`. А если сравниваемые имена ниндзя равны, то возвращается нулевое значение. Обратите внимание на то, что для сравнения имен двух ниндзя можно употреблять простые операции "меньше" (**<**) и "больше" (**>**).

Вот, собственно, и все! Остальные подробности сортировки мы оставим интерпретатору JavaScript, избавляющему нас от ответственности за них.

Агрегирование элементов массива

Сколько раз вам приходилось писать код, подобный приведенному ниже?

```
const numbers = [1, 2, 3, 4];
const sum = 0;

numbers.forEach(number => {
  sum += number;
});

assert(sum === 10, "The sum of first four numbers is 10");
```

В данном коде перебираются все элементы массива и вычисляется некоторое значение, по существу, сводящее весь массив к единственному значению. Такой код можно значительно упростить, воспользовавшись встроенным методом `reduce()`, как показано в примере из листинга 9.12.

Листинг 9.12. Агрегирование элементов массива с помощью метода `reduce()`

```
const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((aggregated, number) =>
                           aggregated + number, 0);

assert(sum === 10, "The sum of first four numbers is 10");
```

Вызвать метод `reduce()`, чтобы вычислить единственное значение из массива

Метод `reduce()` работает следующим образом: для каждого элемента массива вызывается функция обратного вызова, которой передаются текущее значение аккумулятора, накопленное в результате всех предыдущих вызовов этой функции, и текущий элемент массива. При первом вызове этой функции значение аккумулятора полагается равным начальному значению (в данном случае нулю), которое указывается в качестве второго параметра метода `reduce()`. Метод `reduce()` вернет значение аккумулятора, которое было получено в результате последнего обратного вызова для последнего элемента массива. Процесс агрегирования элементов массива наглядно показан на рис. 9.11.

Надеемся, что нам удалось убедить вас, что в JavaScript имеется ряд удобных методов обработки массивов, значительно упрощающих вашу задачу и дающих вам возможность сделать свой прикладной код более изящным, не прибегая к банальной организации циклов `for`. Если вы желаете более подробно ознакомиться с этими и другими методами обработки массивов, обратитесь к их описанию, которое можно найти на веб-сайте Mozilla Developer Network по адресу `https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array`.

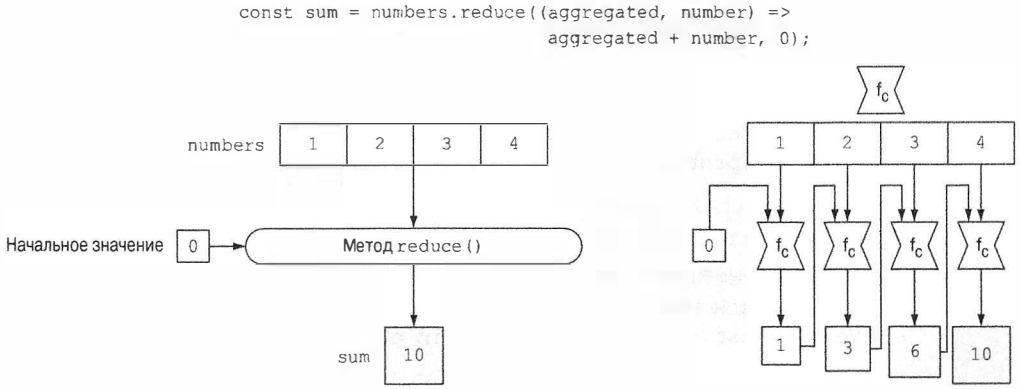


Рис. 9.11. Функции обратного вызова (`fc`) метода **`reduce()`** передается текущее значение аккумулятора и каждый элемент массива, что позволяет свести последний к единственному значению

А теперь пойдем дальше и покажем, как можно использовать упомянутые выше методы обработки массивов по отношению к созданным вами объектам.

9.1.5. Повторное использование встроенных методов обработки массивов

Иногда требуется создать объект, содержащий коллекцию данных. Если наибольший интерес представляет сама коллекция, то для ее обработки можно воспользоваться массивом. Но, помимо коллекции, иногда требуется хранить определенного рода метаданные, связанные с состоянием ее элементов.

С этой целью можно, например, создавать новый массив всякий раз, когда требуется создать новый вариант такого объекта, а также дополнить его методами и свойствами метаданных. Напомним, что объекты, в том числе и массивы, можно дополнять свойствами и методами по своему усмотрению. Но в целом этот процесс может оказаться не только трудоемким, но и медленным.

Рассмотрим возможность снабдить объект требующимися функциональными средствами. Методы обработки коллекций уже существуют для объекта типа `Array`, но можно ли приспособить их к работе со своими объектами? Оказывается, можно, о чем свидетельствует пример кода из листинга 9.13.

Листинг 9.13. Имитация методов обработки массивов

```
<body>
  <input id="first"/>
  <input id="second"/>
  <script>
    const elems = {
      length: 0,
      add: function(elem){
        Array.prototype.push.call(this, elem);
      },
      gather: function(id){
        this.add(document.getElementById(id));
      },
      find: function(callback){
        return Array.prototype.find.call(this, callback);
      }
    };

    elems.gather("first");
    assert(elems.length === 1 && elems[0].nodeType,
           "Verify that we have an element in our stash");

    elems.gather("second");
    assert(elems.length === 2 && elems[1].nodeType,
           "Verify the other insertion");

    elems.find(elem => elem.id === "second");
    assert(found && found.id === "second",
           "We've found our element");
  </script>
</body>
```

Инициализировать счетчик элементов массива. Массиву потребуется место для хранения всего набора его элементов

Реализовать метод, добавляющий элементы в коллекцию. У прототипа `Array` для этого имеется метод, так почему бы не воспользоваться им вместо того, чтобы изобретать колесо?

Реализовать метод-собиратель, в котором элементы HTML-документа находятся по их идентификатору и добавляются в коллекцию

Реализовать метод для поиска элементов в коллекции. Подобно методу `add()`, в нем повторно используется метод `find()`, доступный всем массивам

В данном примере кода сначала создается "обычный" объект, служащий в качестве инструмента для имитации поведения массива. Сначала в нем определяется свойство `length`, в котором отслеживается количество сохраненных элементов, как в массиве, а затем определяется следующий метод `add()` для добавления элемента в конец имитируемого массива:

```
add: function(elem){
  Array.prototype.push.call(this, elem);
}
```

Но вместо того чтобы писать собственный код, можно воспользоваться встроенным в JavaScript методом `Array.prototype.push()`, предназначенным для обработки массивов.

Как правило, метод `Array.prototype.push()` размещает массив в собственном контексте функции. Но в рассматриваемом здесь примере специально используется контекст текущего объекта, который указывается при вызове метода `call()`, хотя для этого можно было бы с тем же успехом вызвать и метод `apply()` (см. главу 4). Метод `push()`, инкрементирующий свойство `length`, считая его настоящим свойством `length` массива, оперирует с числовым свойством объекта, ссылка на который была передана в качестве первого аргумента методу `call()`. Такое поведение носит в известной степени подрывной характер, что очень похоже на ниндзя, но в то же время оно наглядно показывает, как можно манипулировать изменяемыми контекстами объектов.

Метод `add()` ожидает получить ссылку на элемент, передаваемый на хранение. Но чаще всего такая ссылка отсутствует, и поэтому в данном примере определяется удобный метод `gather()`, находящий элемент по значению его идентификатора (`id`) и вводящий его в коллекцию для хранения:

```
gather: function(id){
  this.add(document.getElementById(id));
}
```

И, наконец, в рассматриваемом здесь примере кода определяется метод `find()`, с помощью которого можно найти элемент в специальном объекте, имитирующем поведение массива. И с этой целью в нем вызывается встроенный метод `find()`, предназначенный для обработки массивов:

```
find: function(callback){
  return Array.prototype.find.call(this, callback);
}
```

Столь необычное поведение, продемонстрированное в рассмотренном выше примере кода, не только раскрывает истинный потенциал податливых контекстов функций, но и показывает, как разумно подойти к повторному использованию уже написанного кода, чтобы не изобретать постоянно колесо.

9.2. Отображения

Допустим, в компании `freelanceninja.com` разрабатывается веб-сайт, который требуется сделать доступным для более широкой международной аудитории. С этой целью для каждого фрагмента текста на данном веб-сайте необходимо создать соответствующее отображение на каждом целевом языке (например, японском, китайском или корейском). Такие коллекции, отображающие ключ на конкретное значение, по-разному называются в различных

языках программирования, но чаще всего они известны под названием *словарей* или *отображений*.

Но насколько эффективна подобная локализация в JavaScript? Один из традиционных подходов к решению этой задачи состоит в том, чтобы выгодно воспользоваться объектами как коллекциями именованных свойств и значений, создав словарь, аналогичный следующему:

```
const dictionary = { "ja": {
  "Ninjas for hire": " レンタル用の忍者 "
  },
  "zh": {
    "Ninjas for hire": " 忍者出租 "
  },
  "ko": {
    "Ninjas for hire":"고용 닌자 "
  }
}
assert(dictionary.ja["Ninjas for hire"] === " レンタル用の忍者 ");
            "We know how to say 'Ninjas for hire' in Japanese!");
```

На первый взгляд, такой подход к решению данной задачи может показаться вполне пригодным, и для данного примера он не так уж и плох. Но в общем на него, к сожалению, полагаться нельзя.

9.2.1. Объекты непригодны в качестве отображений

Допустим, что где-нибудь на веб-сайте требуется доступ к переводу слова *constructor*. С этой целью придется расширить приведенный выше пример словаря кодом, выделенным полужирным в листинге 9.14.

Листинг 9.14. Объектам доступны свойства, не определенные явным образом

```
const dictionary = {
  "ja": {
    "Ninjas for hire": " レンタル用の忍者 "
  },
  "zh": {
    "Ninjas for hire": " 忍者出租 "
  },
  "ko": {
    "Ninjas for hire":" 고용 닌자 "

  }
};
assert(dictionary.ja["Ninjas for hire"] === " レンタル用の忍者 ",
      "We know how to say 'Ninjas for hire' in Japanese!");

assert(typeof dictionary.ja["constructor"] === "undefined",
       dictionary.ja["constructor"]);
```

В данном примере кода предпринимается попытка получить доступ к переводу слова *constructor*, которое по забывчивости не было определено в нашем словаре. Как правило, в подобном случае предполагается, что словарь возвратит неопределенное значение (`undefined`). Но на самом деле получается совсем другой результат (рис. 9.12).

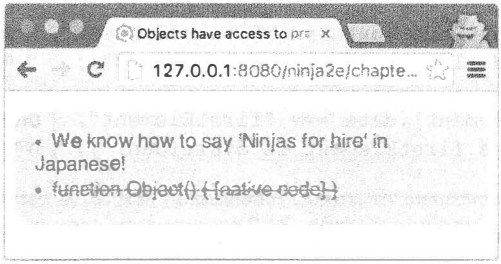


Рис. 9.12. Как показывает результат выполнения кода из листинга 9.14, объекты непригодны в качестве отображений, поскольку им доступны (через их прототипы) свойства, которые не были определены явным образом

Как видите, в результате доступа к свойству `constructor` получается следующая строка кода:

```
"function Object() { [native code] }"
```

Что же это означает? Как пояснялось в главе 7, у всех объектов имеются прототипы. Даже если определить новые пустые объекты в качестве отображений, им все равно будут доступны свойства их прототипов. Одним из них является свойство `constructor` (напомним, что это свойство объекта-прототипа, указывающее обратно на функцию-конструктор). Именно оно и является виновником полученного до сих пор результата выполнения рассматриваемого здесь кода.

Кроме того, ключи в объектах могут содержать только строковые значения. Если же требуется создать отображение на любое другое значение, то это значение будет негласно преобразовано в строковое автоматически! Допустим, требуется отслеживать некоторые сведения об узлах HTML-документа, как демонстрируется в примере кода из листинга 9.15.

Листинг 9.15. Отображение значений на узлы HTML-документа с помощью объектов

Определить два элемента разметки HTML-документа и извлечь их с помощью встроенного метода `getElementById()`

```
<div id="firstElement"></div>
<div id="secondElement"></div>
<script>
  const firstElement = document.getElementById("firstElement");
  const secondElement = document.getElementById("secondElement");
```

```
  const map = {};
  map[firstElement] = { data: "firstElement"};
  assert(map[firstElement].data === "firstElement",
         "The first element is correctly mapped");

  map[secondElement] = { data: "secondElement"};
  assert(map[secondElement].data === "secondElement",
         "The second element is correctly mapped");

  assert(map[firstElement].data === "firstElement",
         "But now the firstElement is overriden!");
</script>
```

Определить объект, предназначенный в качестве отображения для хранения дополнительных сведений об элементах разметки HTML-документа

Сохранить сведения о первом элементе и проверить, правильно ли он был сохранен

Сохранить сведения о втором элементе и проверить, правильно ли он был сохранен

Отображение первого элемента теперь недостоверно!

В примере кода из листинга 9.15 сначала создаются два элемента разметки HTML-документа, `firstElement` и `secondElement`, которые затем извлекаются из модели DOM с помощью метода `document.getElementById()`. Чтобы создать отображение, в котором предполагается хранить дополнительные сведения о каждом элементе, мы определили старый простой объект JavaScript:

```
const map = {};
```

Затем каждый элемент разметки HTML-документа используется в качестве ключа для данного объекта отображения. Некоторые данные связываются с ним следующим образом:

```
map[firstElement] = { data: "firstElement"}
```

А далее проверяется возможность извлечь эти данные. И поскольку эта операция выполняется должным образом, то ее можно полностью повторить и для второго элемента:

```
map[secondElement] = { data: "secondElement"};
```

И в этой операции данные успешно связываются с элементом разметки HTML-документа. Но если затем попытаться снова обратиться к первому элементу, как показано ниже, то возникнет затруднение.

```
map[firstElement].data
```

Казалось бы, мы снова получим сведения о первом элементе, но на самом деле этого не происходит. Вместо этого возвращаются сведения о втором элементе (рис. 9.13).

Это происходит потому, что ключи, представленные объектами, сохраняются в виде символьных строк. И это означает, что при попытке воспользоваться нестроковым значением (например, элементом разметки HTML-документа) как свойством объекта данное значение преобразуется в символьную строку неявно вызываемым методом `toString()`. В данном случае возвращается символьная строка `"[object HTMLDivElement]"`, а сведения о первом элементе сохраняются в виде значения свойства `[object HTMLDivElement]`.

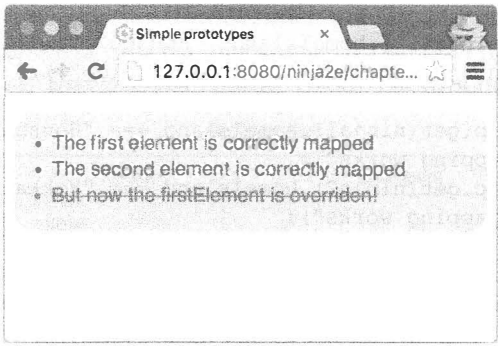


Рис. 9.13. Как показывает результат выполнения кода из листинга 9.15, объекты преобразуются в символьные строки при попытке воспользоваться ими в качестве свойств объекта

То же самое происходит далее при попытке создать отображение для второго элемента. И в этом случае второй элемент, также являющийся элементом `div` разметки HTML-документа, преобразуется в символьную строку, а дополнительные сведения присваиваются свойству `[object HTMLDivElement]`, переопределяя значение, установленное для первого элемента.

Вследствие того что свойства наследуются через прототипы и поддерживаются только строковые ключи, простые объекты, как правило, непригодны в качестве отображений. В силу этих двух ограничений комитет ECMAScript, отвечающий за стандартизацию JavaScript, решил ввести в стандарт *отображения* как совершенно новый тип коллекций.

Примечание

Отображения относятся к стандарту ES6. Их текущую поддержку в браузерах см. по адресу **`http://kangax.github.io/compat-table/es6/#test-Map`**.

9.2.2. Создание первого отображения

Создать отображение совсем не трудно. Для этой цели служит встроенный конструктор объектов типа `Map`. Рассмотрим в качестве примера код из листинга 9.16.

Листинг 9.16. Создание первого отображения

```
const ninjaIslandMap = new Map();
const ninja1 = { name: "Yoshi"};
const ninja2 = { name: "Hattori"};
const ninja3 = { name: "Kuma"};
```

Создать отображение с помощью конструктора объектов типа `Map`

Определить три объекта с именами ниндзя

```
ninjaIslandMap.set(ninja1, { homeIsland: "Honshu"});
ninjaIslandMap.set(ninja2, { homeIsland: "Hokkaido"});
```

Создать отображение для первых двух объектов ниндзя, используя метод отображения `set()`

```
assert(ninjaIslandMap.get(ninja1).homeIsland === "Honshu",
       "The first mapping works");
assert(ninjaIslandMap.get(ninja2).homeIsland === "Hokkaido",
       "The second mapping works");
```

Получить отображение для первых двух объектов ниндзя используя метод отображения `get()`

```
assert(ninjaIslandMap.get(ninja3) === undefined,
       "There is no mapping for the third ninja!");
```

Проверить, что для третьего объекта ниндзя отсутствует отображение

```
assert(ninjaIslandMap.size === 2,
       "We've created two mappings");
```

Проверить наличие отображений для первых двух объектов ниндзя, но не для третьего!

```
assert(ninjaIslandMap.has(ninja1)
       && ninjaIslandMap.has(ninja2),
      "We have mappings for the first two ninjas");
assert(!ninjaIslandMap.has(ninja3),
       "But not for the third ninja!");
```

Проверить с помощью метода `has()`, существует ли отображение для конкретного ключа

```
ninjaIslandMap.delete(ninja1);
assert(!ninjaIslandMap.has(ninja1)
       && ninjaIslandMap.size() === 1,
       "There's no first ninja mapping anymore!");
```

Удалить ключ из отображения с помощью метода `delete()`

```
ninjaIslandMap.clear();
assert(ninjaIslandMap.size === 0,
       "All mappings have been cleared");
```

Полностью очистить отображение с помощью метода `clear()`

Сначала в данном примере кода создается новое отображение, для чего вызывается встроенный конструктор объектов типа `Map`:

```
const ninjaIslandMap = new Map();
```

Затем создаются три объекта, представляющих ниндзя и соответственно названных `ninja1`, `ninja2` и `ninja3`. С этой целью встроенный метод отображения `set()` вызывается таким образом:

```
ninjaIslandMap.set(ninja1, { homeIsland: "Honshu"});
```

В итоге создается отображение ключа (в данном случае объекта `ninja1`) на значение (в данном случае объекта, несущего информацию о родном острове ниндзя). И это делается для первых двух объектов, `ninja1` и `ninja2`, представляющих ниндзя.

Далее получается отображение для первых двух объектов ниндзя с помощью встроенного метода отображения `get()`:

```
assert(ninjaIslandMap.get(ninja1).homeIsland === "Honshu",
       "The first mapping works");
```

Таким образом, отображение существует для первых двух объектов ниндзя, но не для третьего объекта ниндзя, поскольку он не был передан в качестве аргумента методу `set()`. Текущее состояние отображения показано на рис. 9.14.

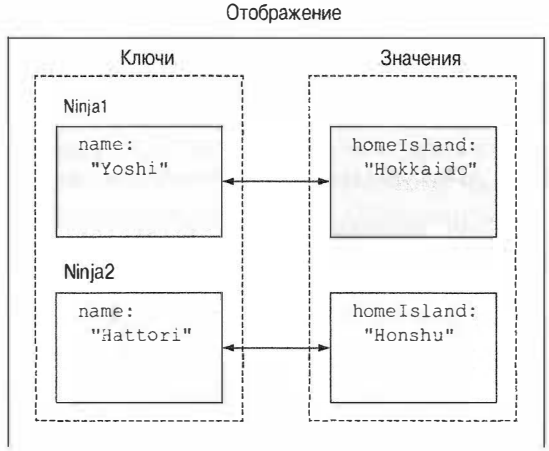


Рис. 9.14. Отображение является коллекцией пар "ключ–значение", где ключом может быть что угодно — даже другой объект

Помимо методов `get()` и `set()`, у каждого отображения имеется также встроенное свойство `size` и методы `has()` и `delete()`. Свойство `size` сообщает, сколько отображений создано. В данном случае создано только два отображения.

С другой стороны, метод `has()` извещает, существует ли отображение для конкретного ключа:

```
ninjaIslandMap.has(ninja1); // истина
ninjaIslandMap.has(ninja3); // ложь
```

А метод `delete()` позволяет удалить элементы из отображения следующим образом:

```
ninjaIslandMap.delete(ninja1);
```

Одним из основных принципов работы с отображениями является определение момента, когда ключи отображения оказываются равными. Рассмотрим этот принцип более подробно.

Равенство ключей

Если у вас имеется опыт программирования на таких традиционных языках, как C#, Java или Python, то вас может удивить код из примера, демонстрируемого в листинге 9.17.

Листинг 9.17. Равенство ключей в отображениях

```
const map = new Map();
const currentLocation = location.href;

const firstLink = new URL(currentLocation);
const secondLink = new URL(currentLocation);

map.set(firstLink, { description: "firstLink"});
map.set(secondLink, { description: "secondLink"});

assert(map.get(firstLink).description === "firstLink",
       "First link mapping" );
assert(map.get(secondLink).description === "secondLink",
       "Second link mapping");
assert(map.size === 2, "There are two mappings");
```

Воспользоваться встроенным свойством `location.href` для получения URL текущей веб-страницы

Создать две ссылки на текущую страницу

Создать отображение для обеих ссылок

Для каждой ссылки создается свое отображение, хотя они и указывают на одну и ту же страницу

Сначала в коде примера из листинга 9.17 встроенное свойство `location.href` применяется для получения URL текущей веб-страницы. Затем с помощью встроенного конструктора в данном коде создаются два новых объекта типа `URL`, обеспечивающих ссылки на текущую страницу. После этого каждая ссылка связывается с соответствующим объектом ее описания. Наконец, в данном коде проверяется правильность созданных отображений (рис. 9.15).

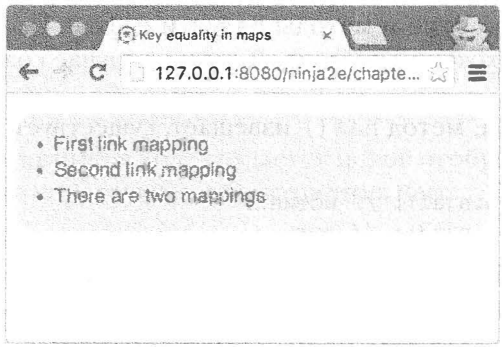


Рис. 9.15. Если выполнить код из листинга 9.17, то можно заметить, что равенство ключей в отображении зависит от равенства объектов

Даже тем, кто программировал в основном на JavaScript, такой результат может показаться неожиданным. Ведь в данном случае имеются два объекта, для которых создаются два разных отображения. Но обратите внимание на то, что оба объекта типа `URL` по-прежнему указывают на одно и то же местонахождение текущей веб-страницы, хотя это и совершенно разные объекты. Можно было бы утверждать, что при создании отображений оба эти объекта следует считать равными. Но в языке JavaScript не допускается перегрузка операции равенства, и поэтому оба объекта всегда считаются разными, даже если у них одинаковое

содержимое. Совсем иначе дело обстоит в других языках, в том числе Java и C#, поэтому будьте внимательны!

9.2.3. Перебор элементов отображений

Выше были описаны некоторые преимущества отображений. В частности, вы могли убедиться, что они содержат только те элементы, которые в них были добавлены, а в качестве ключа можно использовать все, что угодно. Но имеется немало других преимуществ!

Отображения являются коллекциями, и поэтому ничто не мешает выполнить перебор их элементов в циклах `for-of`. (Напомним, что цикл `for-of` применялся в примерах из главы 6 для перебора значений, создаваемых генераторами.) При этом также гарантируется, что значения будут выбираться в том порядке, в каком они добавлялись, чего нельзя гарантировать при переборе свойств объектов в цикле `for-in`. Рассмотрим в качестве примера код из листинга 9.18.

Листинг 9.18. Перебор элементов отображений

```
const directory = new Map();
```

Создать новое отображение так же, как мы это делали раньше

```
directory.set("Yoshi", "+81 26 6462");
directory.set("Kuma", "+81 52 2378 6462");
directory.set("Hiro", "+81 76 277 46");
```

Создать справочник ниндзя с номерами их телефонов

```
for(let item of directory){
  assert(item[0] !== null, "Key:" + item[0]);
  assert(item[1] !== null, "Value:" + item[1]);
}
```

Перебрать каждый элемент справочника в цикле `for-of`. Каждый элемент представляет собой двухэлементный массив, содержащий ключ и значение

```
for(let key of directory.keys()){
  assert(key !== null, "Key:" + key);
  assert(directory.get(key) != null,
         "Value:" + directory.get(key));
}
```

Ключи можно также перебрать с помощью встроенного метода `keys()`...

```
for(var value of directory.values()){
  assert(value !== null, "Value:" + value);
}
```

...а значения — с помощью встроенного метода `values()`

Как следует из примера кода в листинге 9.18, создав отображение, можно без особого труда перебрать все его элементы цикле `for-of`, как показано ниже.

```
for(var item of directory){
  assert(item[0] !== null, "Key:" + item[0]);
  assert(item[1] !== null, "Value:" + item[1]);
}
```

На каждом шаге приведенного выше цикла мы имеем дело с двухэлементным массивом, где ключ помещен в первый его элемент, а значение, получаемое из отображения справочника – во второй. Для перебора всех ключей и значений, содержащихся в отображении, можно также воспользоваться методами `keys()` и `values()`.

А теперь перейдем к обсуждению еще одного нововведения в JavaScript: *множеств*, являющихся коллекциями уникальных элементов.

9.3. Множества

При решении многих практических задач приходится иметь дело с коллекциями *индивидуальных* элементов, называемых *множествами*, где все элементы оказываются разными. До введения стандарта ES6 некоторые имитации множеств приходилось реализовывать с помощью обычных объектов. В качестве примера грубой имитации множеств рассмотрим код из листинга 9.19.

Листинг 9.19. Имитация множеств с помощью объектов

```
function Set(){
  this.data = {};
  this.length = 0;
}

Set.prototype.has = function(item){
  return typeof this.data[item] !== "undefined";
};

Set.prototype.add = function(item){
  if(!this.has(item)){
    this.data[item] = true;
    this.length++;
  }
};

Set.prototype.remove = function(item){
  if(this.has(item)){
    delete this.data[item];
    this.length--;
  }
};

const ninjas = new Set();
ninjas.add("Hattori");
ninjas.add("Hattori");

assert(ninjas.has("Hattori") && ninjas.length === 1,
       "Our set contains only one Hattori");
```

Воспользоваться объектом для хранения элементов множества

Проверить, сохранен ли уже элемент в множестве

Добавить элемент только в том случае, если он отсутствует в множестве

Удалить элемент, если он присутствует в множестве

Попытаться добавить элемент "Hattori" дважды

Проверить, был ли элемент "Hattori" добавлен в множество только один раз

```
ninjas.remove("Hattori");
assert(!ninjas.has("Hattori") && ninjas.length === 0,
       "Our set is now empty");
```

Удалить элемент "Hattori" и проверить, действительно ли он удален из множества

В примере кода из листинга 9.19 показано, каким образом множества можно сымитировать с помощью объектов. В частности, объект информационного хранилища `data` служит для хранения элементов множества, а для обработки множества определяются три метода. Так, в методе `has()` проверяется, существует ли уже элемент в множестве. Метод `add()` добавляет элемент в множество только в том случае, если он отсутствует в нем. А метод `remove()` удаляет элемент, уже существующий в множестве.

Но это довольно грубая имитация множества, поскольку она позволяет хранить только символьные строки и числа, но не объекты. А кроме того, всегда существует риск доступа к свойствам объектов-прототипов. Именно по этим причинам комитет ECMAScript, отвечающий за стандартизацию языка JavaScript, решил внедрить *множества* как совершенно новый тип коллекций.

Примечание

Множества относятся к стандарту ES6 языка JavaScript. Их текущую поддержку в браузерах см. по адресу **`http://kangax.github.io/compat-table/es6/#test-Set`**.

9.3.1. Создание первого множества

Краеугольным камнем создания множеств служит новая функция-конструктор объектов типа `Set`. Рассмотрим создание множества на примере кода из листинга 9.20.

Листинг 9.20. Создание множества

Конструктору объектов типа Set может передаваться массив, предназначенный для инициализации элементов множества

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);
```

Отбросить любые дубликаты элементов множества

```
assert(ninjas.has("Hattori"), "Hattori is in our set");
assert(ninjas.size === 3, "There are only three ninjas in our set!");

assert(!ninjas.has("Yoshi"), "Yoshi is not in, yet..");
ninjas.add("Yoshi");
assert(ninjas.has("Yoshi"), "Yoshi is added");
assert(ninjas.size === 4, "There are four ninjas in our set!");

assert(ninjas.has("Kuma"), "Kuma is already added");
ninjas.add("Kuma");
assert(ninjas.size === 4, "Adding Kuma again has no effect");
```

Добавить новые элементы, еще отсутствующие в множестве

Попытка добавить элементы, уже существующие в множестве, ничего не дает

```
for(let ninja of ninjas) {
  assert(ninja !== null, ninja);
}
```

Перебрать элементы множества в цикле `for-of`

В данном примере кода новое множество `ninjas`, которое будет содержать индивидуальные имена ниндзя, создается с помощью встроенного конструктора объектов типа `Set`. Если не передать этому конструктору никаких аргументов, то будет создано пустое множество. Кроме того, данному конструктору можно передать массив, предназначенный для предварительного заполнения множества, как показано в следующей строке кода:

```
new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);
```

Как упоминалось ранее, множества являются коллекциями уникальных элементов, а их основное назначение — исключить сохранение дубликатов экземпляров одного и того же объекта. В данном случае это означает, что при попытке дважды сохранить символьную строку `"Hattori"` она добавляется в множество только один раз.

Для каждого множества доступен целый ряд методов. Например, в методе `has()` проверяется, содержится ли указанный элемент в множестве:

```
ninjas.has("Hattori")
```

Метод `add()` служит для добавления уникальных элементов в множество:

```
ninjas.add("Yoshi");
```

Если же требуется выяснить, сколько элементов содержится в множестве, то для этой цели можно всегда воспользоваться свойством `size`.

Подобно отображениям и массивам, множества являются коллекциями, а следовательно, их элементы можно перебирать в цикле `for-of`. Как показано на рис. 9.16, элементы всегда перебираются в том порядке, в каком они были добавлены в множество.

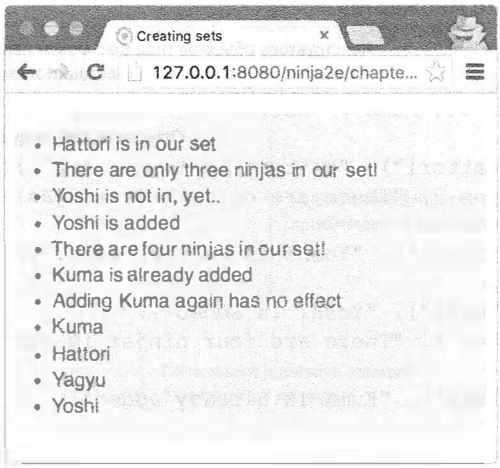


Рис. 9.16. Как показывает выполнение кода из листинга 9.20, элементы перебираются в том порядке, в каком они были добавлены в множество

Итак, рассмотрев основы организации множеств, обсудим некоторые наиболее употребительные операции над множествами: объединения, пересечения и разности.

9.3.2. Объединение множеств

В результате объединения множеств `A` и `B` образуется новое множество, содержащее уникальные элементы обоих исходных множеств: `A` и `B`. По определению, каждый элемент может присутствовать в новом множестве только один раз. В коде из листинга 9.21 демонстрируется наглядный пример объединения двух множеств.

Листинг 9.21. Объединение множеств с целью слияния элементов коллекций данных

```
const ninjas = ["Kuma", "Hattori", "Yagyu"];
const samurai = ["Hattori", "Oda", "Tomoe"];

const warriors = new Set([...ninjas, ...samurai]);

assert(warriors.has("Kuma"), "Kuma is here");
assert(warriors.has("Hattori"), "And Hattori");
assert(warriors.has("Yagyu"), "And Yagyu");
assert(warriors.has("Oda"), "And Oda");
assert(warriors.has("Tomoe"), "Tomoe, last but not least");

assert(warriors.size === 5, "There are 5 warriors in total");
```

Создать массивы `ninjas` и `samurai`. Обратите внимание, что имя `Hattori` носит как ниндзя, так и самурай

Создать новое множество воинов из двух исходных множеств ниндзя и самураев

Все ниндзя и самураи включены в новое множество воинов

Дубликаты в новом множестве отсутствуют. И хотя имя Hattori присутствует в обоих множествах ниндзя и самураев, в новое множество оно включено только один раз

Сначала в данном примере кода создаются массивы `ninjas` и `samurai`. Обратите внимание на то, что воин по имени `Hattori` ведет довольно активный образ жизни: днем он самурай, а ночью — ниндзя. А теперь допустим, что требуется создать коллекцию людей, которых можно призвать к оружию, если соседний даймё (т.е. местный феодал) решит, что его владения следовало бы немного расширить. Далее в данном примере создается новое множество `warriors`, включающее в себя имена всех ниндзя и самураев. И хотя имя `Hattori` присутствует в обеих исходных коллекциях, в новую коллекцию оно будет включено только один раз. Таким образом, на призыв местного феодала откликнется только один воин по имени `Hattori`.

В данном случае объединенное множество выглядит идеально! Нам не нужно отслеживать вручную, находится ли элемент уже в множестве, поскольку эта операция выполняется множеством автоматически. Для создания нового множества применяется операция расширения `[...ninjas, ...samurai]` (см. главу 3), с помощью которой образуется новый массив, содержащий имена всех ниндзя и самураев. Любопытно, что имя `Hattori` присутствует в новом массиве дважды. Но когда этот массив наконец передается конструктору объектов типа

Set, в объединенное множество имя Hattori включается только один раз, как показано на рис. 9.17.

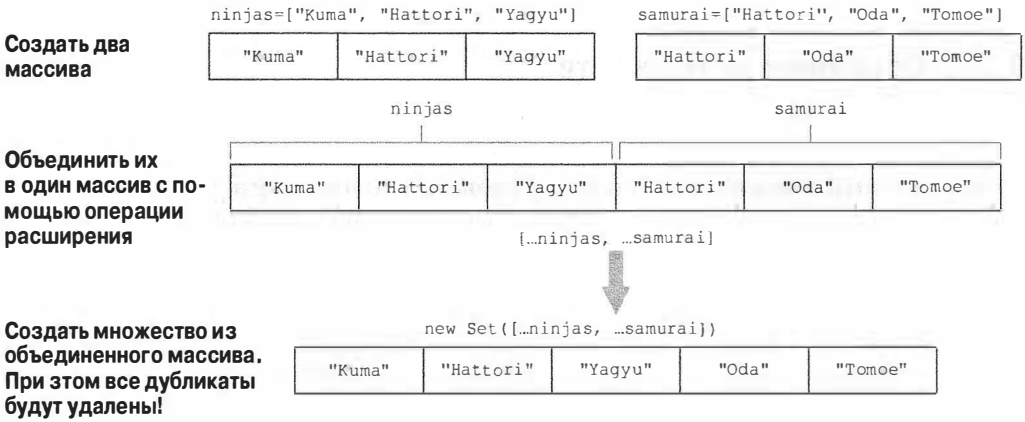


Рис. 9.17. В объединенных множествах сохраняются элементы из исходных коллекций, но без дубликатов

9.3.3. Пересечение множеств

При пересечении двух множеств, A и B, создается новое множество, содержащее элементы, общие для обоих исходных множеств A и B. Так, в примере кода из листинга 9.22 демонстрируется, каким образом можно найти ниндзя, которые являются также самураями.

Листинг 9.22. Пересечение множеств

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
const samurai = new Set(["Hattori", "Oda", "Tomoe"]);

const ninjaSamurais = new Set(
  [...ninjas].filter(ninja => samurai.has(ninja))
);

assert(ninjaSamurais.size === 1, "There's only one ninja samurai");
assert(ninjaSamurais.has("Hattori"), "Hattori is his name");
```

Превратить множество в массив с помощью операции расширения, а затем вызвать метод `filter()`, чтобы сохранить в новом множестве только тех ниндзя, которые присутствуют и в множестве `samurai`

Общий замысел кода из листинга 9.22 состоит в том, чтобы создать новое множество, содержащее только тех ниндзя, которые являются также самураями. С этой целью выгодно используется метод filter() обработки массивов, в котором, как упоминалось ранее, создается новый массив, содержащий только те элементы, которые удовлетворяют определенному критерию. В данном случае критерий заключается в том, что ниндзя должны быть также самураями, т.е. присутствуют не только в множестве ninjas, но и в множестве samurai. А поскольку метод filter() можно применять только к массивам, то множество ninjas преобразуется в массив с помощью следующей операции расширения:

```
[...ninjas]
```

И, наконец, в рассматриваемом здесь примере кода проверяется, что в новом множестве находится только один ниндзя, который является также самураем. И этого мастера на все руки зовут `Hattori`.

9.3.4. Разность множеств

Разность двух множеств `A` и `B` содержит все элементы, имеющиеся во множестве `A`, но отсутствующие во множестве `B`. Нетрудно догадаться, что операция разности похожа на операцию пересечения, за исключением одной небольшой, но очень существенной разницы. В примере кода из листинга 9.23 демонстрируется, как выявить настоящих ниндзя, которые не служат по совместительству самураями.

Листинг 9.23. Разность множеств

```
const ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
const samurai = new Set(["Hattori", "Oda", "Tomoe"]);

const pureNinjas = new Set(
  [...ninjas].filter(ninja => !samurai.has(ninja))
);

assert(pureNinjas.size === 2, "There's only one ninja samurai");
assert(pureNinjas.has("Kuma"), "Kuma is a true ninja");
assert(pureNinjas.has("Yagyu"), "Yagyu is a true ninja");
```

Найдя разность множеств, можно выявить тех ниндзя, которые не являются самураями!

Чтобы выявить тех ниндзя, которые не являются самураями, в данном коде достаточно было поставить знак восклицания (`!`) перед выражением `samurai.has(ninja)` по сравнению с кодом из листинга 9.22.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Массивы являются специальным типом объектов, в которых поддерживаются свойство `length` и прототип `Array.prototype`.
- Новые массивы можно создавать с помощью литерала массива (`[]`) или путем вызова встроенного конструктора объектов типа `Array`.
- Содержимое массива можно видоизменить, используя ряд перечисленных ниже методов, поддерживаемых в объектах массивов.
  - Встроенные методы `push()` и `pop()`, добавляющие и удаляющие элементы в конце массива.
  - Встроенные методы `shift()` и `unshift()`, добавляющие и удаляющие элементы в начале массива.
  - Встроенный метод `splice()`, добавляющий и удаляющий элементы в произвольных местах массива.

- Для всех массивов поддерживаются следующие полезные методы.
  - Метод `map()`, создающий новый массив на основе возвращаемых значений из функции обратного вызова для каждого элемента.
  - Методы `every()` и `some()`, определяющие, удовлетворяют ли определенному критерию все или некоторые элементы массива.
  - Методы `find()` и `filter()`, выявляющие элементы массива, удовлетворяющие определенному критерию.
  - Метод `sort()`, сортирующий массив.
  - Метод `reduce()`, агрегирующий все элементы массива в единственное значение.
- Встроенные методы обработки массивов можно использовать повторно при реализации собственных объектов, явно задавая контекст вызова конкретного метода с помощью метода `call()` или `apply()`.
- Отображения и словари являются объектами, содержащими наборы ключей и соответствующих им значений.
- Объекты в JavaScript непригодны для создания отображений, поскольку в качестве ключей могут служить только строковые значения и всегда существует риск доступа к свойствам прототипов. Вместо этого для создания отображений лучше воспользоваться встроенной коллекцией типа `Map`.
- Отображения являются коллекциями, и поэтому их элементы можно перебирать в цикле `for-of`.
- Множества являются коллекциями уникальных элементов.

Упражнения

1. Каким окажется содержимое массива `samurai` после выполнения следующего фрагмента кода?

```
const samurai = ["Oda", "Tomoe"];
samurai[3] = "Hattori";
```

2. Каким окажется содержимое массива `ninjas` после выполнения следующего фрагмента кода?

```
const ninjas = [];

ninjas.push("Yoshi");

ninjas.unshift("Hattori");

ninjas.length = 3;

ninjas.pop();
```

3. Каким окажется содержимое массива samurai после выполнения следующего фрагмента кода?

```
const samurai = [];

samurai.push("Oda");
samurai.unshift("Tomoe");
samurai.splice(1, 0, "Hattori", "Takeda");
samurai.pop();
```

4. Что останется в переменных first, second и third после выполнения следующего фрагмента кода?

```
const ninjas = [{name:"Yoshi", age: 18},
                {name:"Hattori", age: 19},
                {name:"Yagyu", age: 20}];

const first = ninjas.map(ninja => ninja.age);
const second = first.filter(age => age % 2 == 0);
const third = first.reduce((aggregate, item) => aggregate + item, 0);
```

5. Что останется в переменных first и second после выполнения следующего фрагмента кода?

```
const ninjas = [{ name: "Yoshi", age: 18 },
                { name: "Hattori", age: 19 },
                { name: "Yagyu", age: 20 }];

const first = ninjas.some(ninja => ninja.age % 2 == 0);
const second = ninjas.every(ninja => ninja.age % 2 == 0);
```

6. Какие из приведенных ниже утверждений пройдут?

```
const samuraiClanMap = new Map();

const samurai1 = { name: "Toyotomi"};
const samurai2 = { name: "Takeda"};
const samurai3 = { name: "Akiyama"};

const oda = { clan: "Oda"};
const tokugawa = { clan: "Tokugawa"};
const takeda ={clan: "Takeda"};

samuraiClanMap.set(samurai1, oda);
samuraiClanMap.set(samurai2, tokugawa);
samuraiClanMap.set(samurai2, takeda);

assert(samuraiClanMap.size === 3,
       "There are three mappings");
assert(samuraiClanMap.has(samurai1),
       "The first samurai has a mapping");
assert(samuraiClanMap.has(samurai3),
       "The third samurai has a mapping");
```

7. Какие из приведенных ниже утверждений пройдут?

```
const samurai = new Set("Toyotomi", "Takeda", "Akiyama", "Akiyama");

assert(samurai.size === 4, "There are four samurai in the set");

samurai.add("Akiyama");
assert(samurai.size === 5, "There are five samurai in the set");

assert(samurai.has("Toyotomi", "Toyotomi is in!");
assert(samurai.has("Hattori", "Hattori is in!");
```

10 Овладение регулярными выражениями

В этой главе...

- Основные сведения о регулярных выражениях
- Компилирование регулярных выражений
- Фиксация результатов с помощью регулярных выражений
- Решение часто встречающихся задач с помощью регулярных выражений

Без регулярных выражений трудно обойтись в разработке современного программного обеспечения. И хотя многие разработчики могут счастливо прожить, не прибегая к регулярным выражениям, некоторые задачи программирования на JavaScript без них изящно решить нельзя.

Безусловно, одни и те же задачи можно решать по-разному. Но зачастую полстраницы кода можно заменить одной строкой, применяя надлежащим образом регулярные выражения. Поэтому всякий, стремящийся стать настоящим мастером программирования на JavaScript, должен включить регулярные выражения в арсенал своих средств.

Регулярные выражения упрощают процесс разбиения символьных строк на отдельные части и поиска в них информации. В какую бы из основных библиотек JavaScript ни заглянуть, везде можно обнаружить, что регулярным выражениям отводится главенствующая роль в решении следующих задач.

- Манипулирование строками HTML-документа.
- Обнаружение неполных селекторов в выражениях с использованием CSS-селекторов.

- Определение имени конкретного класса у элемента.
- Проверка достоверности вводимых данных.
- И многое другое...

Совет

Для овладения регулярными выражениями требуется немалая практика. Оперативно поупражняться на примерах регулярных выражений можно, например, на веб-сайте *JS Bin* (`http://jsbin.com/?html,output`). Два других веб-сайта, Test Page for JavaScript (`http://www.regexplanet.com/advanced/javascript/index.html`) и regex101 (`https://www.regex101.com/#javascript`), посвящены проверке регулярных выражений в оперативном режиме. Особенно полезен для начинающих веб-сайт regex101, поскольку на нем автоматически формируются пояснения целевых регулярных выражений.

Знаете ли вы?

Когда следует отдавать предпочтение литералу над объектом регулярного выражения?

Что такое "липкое" совпадение и как оно активизируется?

Чем отличается совпадение при использовании глобального и неглобального регулярного выражения?

Итак, начнем рассмотрение регулярных выражений с простого примера.

10.1. Достоинства регулярных выражений

Допустим, требуется убедиться, что символьная строка, введенная в форме, заполняемой посетителем веб-сайта, соответствует принятому в США формату почтового кода, состоящему из девяти цифр. Всем жителям США известно, что почтовое ведомство этой страны не любит шуток и категорически настаивает на том, чтобы почтовый код (или так называемый почтовый индекс) соответствовал следующему конкретному формату:

```
99999-9999
```

где `9` — десятичная цифра. Этот формат состоит из пяти десятичных цифр, дефиса и еще четырех десятичных цифр. Если указать на конверте с письмом или упаковке с посылкой почтовый индекс в другом формате, такое почтовое отправление затеряется где-то в недрах отделения ручной сортировки почты, и никто не знает, когда оно снова появится на свет.

Мы создадим функцию, которая будет проверять переданную ей символьную строку на соответствие формату почтового индекса, принятому в США. Для этого можно было бы прибегнуть к сравнению каждого символа в строке, но вряд ли настоящий мастер посчитает такое решение изящным, поскольку оно подразумевает частое и ненужное повторение кода. Вместо этого рассмотрим решение, приведенное в примере кода из листинга 10.1.

Листинг 10.1. Проверка символьной строки по конкретному шаблону

```
function isThisAZipCode(candidate) {
  if (typeof candidate !== "string" ||
      candidate.length != 10) return false;
  for (let n = 0; n < candidate.length; n++) {
    let c = candidate[n];
    switch (n) {
      case 0: case 1: case 2: case 3: case 4:
      case 6: case 7: case 8: case 9:
        if (c < '0' || c > '9') return false;
        break;
      case 5:
        if (c != '-') return false;
        break;
    }
  }
  return true;
}
```

Другие типы и короткие строки, очевидно, не подходят

Выполнить проверку по индексу символа в строке

Если все прошло успешно, то и хорошо!

В приведенном выше примере кода выгодно используется тот факт, что в зависимости от положения символа в строке производятся только две разные проверки. А во время выполнения кода по-прежнему приходится выполнять до девяти сравнений, хотя код для каждого сравнения достаточно написать лишь один раз. Но следует ли считать такое решение изящным? Безусловно, оно более изящно, чем грубый неитерационный подход, но все-таки оно предполагает написание слишком большого объема кода для реализации столь простой проверки.

А теперь рассмотрим еще одно решение:

```
function isThisAZipCode(candidate) {
  return /^\d{5}-\d{4}$/.test(candidate);
}
```

Как видите, это намного более изящное и компактное решение, если не считать довольно таинственного вида кода в теле функции. В то же время данный пример наглядно демонстрирует потенциал, скрывающийся в регулярных выражениях, как в подводной части айсберга. Конечно, синтаксис регулярного выражения обычно выглядит так, как будто оно было набрано обезьяной на клавиатуре. Но не смущайтесь — мы сделаем краткий обзор регулярных выражений, прежде чем перейти к их мастерскому применению в разработке веб-приложений на JavaScript.

10.2. Основные сведения о регулярных выражениях

Как бы нам ни хотелось, но мы не можем позволить себе пространное изложение регулярных выражений в силу ограниченности места в книге. Впрочем,

на эту тему имеется довольно обширная литература. В частности, рекомендуем следующие книги: *Mastering Regular Expressions* Джеффри Е.Ф. Фридла (Jeffrey E.F. Friedl; издательство O'Reilly; в русском переводе книга вышла под названием *Регулярные выражения* в издательстве "Символ-Плюс", 2008 г.), *Introducing Regular Expressions* Майкла Фицджеральда (Michael Fitzgerald, издательство O'Reilly; в русском переводе книга вышла под названием *Регулярные выражения: основы* в ИД "Вильямс, 2016 г.), а также книгу Бена Форты *Регулярные выражения за 10 минут* (пер. с англ., ИД "Вильямс, 2017 г.). Можно почитать книгу *Regular Expressions Cookbook* Яна Гойвертца и Стивена Левитана (Jan Goyvaerts, Steven Levithan; издательство O'Reilly; 2012 г.). Но мы все же постараемся ниже осветить все самое главное, что касается регулярных выражений.

10.2.1. Назначение регулярных выражений

Термин *регулярное выражение* появился в математике в середине XX века, когда американский математик Стивен Клини (Stephen Kleene) занимался описанием моделей автоматов (абстрактных вычислительных устройств) как "регулярных множеств". Но это объяснение вряд ли поможет лучше понять назначение регулярных выражений, поэтому сформулируем его проще: регулярное выражение — это способ обозначить *шаблон* для поиска совпадений с текстовыми строками. Само регулярное выражение состоит из членов и операций, позволяющих определять такие шаблоны. Ниже будет показано, что собой представляют члены и операции регулярного выражения.

В языке JavaScript, как и в большинстве других объектно-ориентированных языков программирования, регулярное выражение может быть создано двумя способами:

- с помощью литерала регулярного выражения;
- путем создания экземпляра объекта типа `RegExp`.

Так, если требуется создать очень простое регулярное выражение для точного совпадения с символьной строкой **`"test"`**, это можно сделать с помощью литерала регулярного выражения следующим образом:

```
var pattern = /test/;
```

Такое регулярное выражение с косыми чертами выглядит не совсем обычно, но литералы регулярных выражений заключаются в косые черты точно так же, как и строковые литералы в кавычки. С другой стороны, можно создать экземпляр объекта типа `RegExp`, передав ему регулярное выражение в качестве символьной строки, как показано ниже.

```
var pattern = new RegExp("test");
```

В обоих случаях создается одно и то же регулярное выражение, сохраняемое в переменной `pattern`.

Совет

Если регулярное выражение заранее известно во время разработки, то предпочтение отдается синтаксису литерала, тогда как синтаксис конструктора применяется при динамическом создании регулярного выражения прямо во время выполнения кода.

Одна из причин, по которой следует отдавать предпочтение синтаксису литерала регулярных выражений, а не синтаксису конструктора, состоит в том, что, как станет ясно в дальнейшем, символ обратной косой черты играет очень важную роль в регулярных выражениях. Однако сам символ обратной косой черты также используется для экранирования специальных символов в строковых литералах, и поэтому для обозначения его самого в строковом литерале приходится его указывать в виде двойной обратной косой черты (\\). Именно поэтому регулярные выражения, которые и без того обладают не очень понятным синтаксисом, могут иметь совершенно странный вид, когда они выражаются в виде символьных строк.

Помимо самого выражения, в регулярном выражении могут быть указаны следующие модификаторы:

- **i** – делает регулярное выражение не зависящим от регистра, поэтому регулярное выражение `/test/i` соответствует не только с символьной строке `"test"`, но и строкам `"Test"`, `"TEST"`, `"tEsT"` и т.д.
- **g** – выполняет поиск всех элементов строки, которые соответствуют заданному шаблону, а не только его первого экземпляра, как принято по умолчанию. Подробнее об этом речь пойдет далее.
- **m** – допускает поиск совпадений в нескольких строках текста, которые могут быть получены из элемента управления формы типа `textarea`.
- **y** – активизирует "липкое" сопоставление. При этом сопоставление в целевой строке начинается с последней найденной позиции (точнее с индекса, на который указывает свойство `lastIndex` этого регулярного выражения).
- **u** – позволяет пользоваться экранированием кодовых точек в Юникоде (**\u{...}**).

Эти модификаторы могут добавляться к концу литерала (например, `/test/ig`) или же передаваться в виде символьной строки в качестве второго параметра конструктора объектов типа `RegExp` (например, `new RegExp("test", "ig")`). Простое совпадение с символьной строкой `"test"` (пускай даже точное и без учета регистра) не особенно интересно. Ведь такую проверку можно выполнить простым сравнением символьных строк. Поэтому рассмотрим члены и операции, которые наделяют регулярные выражения огромным потенциалом, позволяющим сопоставлять строки с более сложными шаблонами.

10.2.2. Члены и операции

Регулярные выражения, как и большинство других известных вам выражений, состоят из членов и операций, уточняющих эти члены. В последующих подразделах мы рассмотрим эти члены и операции и покажем, как пользоваться ими для обозначения шаблонов.

Точное соответствие

Любой символ, не являющийся специальным и не обозначающий операцию (операции мы рассмотрим чуть позже), представляет собой символ, который должен буквально присутствовать в выражении. Например, в упоминавшемся выше примере регулярного выражения `/test/` имеются четыре члена, представляющих символы, которые должны буквально присутствовать в символьной строке, чтобы она точно совпадала с шаблоном, указанным в данном регулярном выражении. Расположение символов друг за другом неявно указывает на операцию *следования*. Таким образом, регулярное выражение `/test/` означает, что за символом **t** следует символ `e`, за ним — символ **s**, а затем — символ **t**.

Совпадение с классом символов

Зачастую требуется совпадение не с одним конкретным символом, а с любым символом из конечного набора. Такое условие можно задать с помощью операции над множеством, называемой также *операцией над классом символов*, для чего набор сопоставляемых символов заключается в квадратные скобки. Например, выражение `[abc]` означает, что требуется совпадение с любым из символов **a**, **b** или **c**. Следует, однако, иметь в виду, что совпадение в данном примере будет происходить только с одним символом в проверяемой строке, несмотря на то, что регулярное выражение фактически состоит из пяти символов (две "служебные" скобки и три "рабочих" символа).

Но иногда требуется совпадение со всеми символами, *кроме* указанных в конечном наборе. Для этого достаточно указать знак вставки (**^**) сразу после открывающей квадратной скобки в операции над множеством:

```
[^abc]
```

И в этом случае совершенно меняется смысл регулярного выражения. Теперь оно означает совпадение с любыми символами, *кроме* символов **a**, **b** или **c**.

Имеется еще одна бесценная разновидность операции над множеством, позволяющая указывать диапазон значений. Так, если бы требовалось совпадение с любой строчной буквой в пределах от **a** до **m**, для этой цели можно было бы составить выражение `[abcdefghijklm]`. Но то же самое условие можно выразить более компактно, как показано ниже.

```
[a-m]
```

Дефис в данном выражении обозначает диапазон символов от **a** до **m** включительно, т.е. в сопоставляемый набор символов входят буквально все указанные выше строчные буквы.

Экранирование

Не все символы в регулярном выражении представляют "самих себя" (т.е. свой буквальный эквивалент). Разумеется, все буквенно-цифровые символы представляют самих себя, но, как будет показано ниже, специальные знаки, например, точки (`.`) и денежной единицы (`$`), обозначают совпадение с каким-то другим символом, а не с самими собой, или же операции, уточняющие предыдущий член выражения. В представленных ранее примерах регулярных выражений вам уже встречались знаки `[`, `]`, `-` и `^`, представляющие не самих себя, а нечто другое.

Как же указать, что требуется совпадение с самим специальным знаком, например `[`, `$` или `^`? Для этой цели в регулярном выражении служит знак обратной косой черты, "экранирующий" любой следующий за ним символ, делая его тем самым буквально сопоставляемым членом данного выражения. Следовательно, последовательность символов `\[` обозначает буквальное совпадение со знаком `[`, а не открытие выражения с классом символов, а двойная обратная косая черта (`\\`) — совпадение с единственным знаком обратной косой черты.

Начало и конец сопоставлений

Нередко требуется, чтобы сопоставление с шаблоном происходило в начале, а возможно, и в конце символьной строки. Так, если знак вставки указывается первым символом в регулярном выражении, сопоставление с шаблоном привязывается к началу символьной строки. Например, в регулярном выражении `/^test/` совпадение произойдет лишь в том случае, если подстрока `"test"` окажется в начале проверяемой символьной строки. (Следует заметить, что это своего рода перегрузка знака `^`, поскольку он используется также в классах символов для исключения набора символов из сопоставления.)

Аналогично знак денежной единицы (`$`) обозначает, что сопоставляемый шаблон должен появиться в конце символьной строки: **`/test$/`**. Если же в выражении используются оба знака, `^` и `$`, это означает, что указанный шаблон должен охватывать всю проверяемую символьную строку: `/^test$/`.

Повторяющиеся экземпляры

Если требуется сопоставить строку с четырьмя подряд символами **`a`**, то шаблон для этой операции можно выразить как `/aaaa/`. Но что, если требуется совпадение с *любым* количеством одного и того же символа? Для указания количества различных вариантов повторений в регулярных выражениях предоставляются следующие операции.

- Если требуется задать символ как необязательный (иными словами, он может присутствовать один раз или вообще отсутствовать в проверяемой на совпадение строке), то после него следует указать знак вопроса (`?`). Например, регулярное выражение `/t?est/` обеспечивает совпадение как со строкой `"test"`, так и со строкой `"est"`.

- Если требуется, чтобы символ присутствовал однократно или многократно в проверяемой на совпадение строке, то после него следует указать знак "плюс" (**+**). Например, регулярное выражение `/t+est/` обеспечивает совпадение со строками `"test"`, `"ttest"` и `"tttest"`, но не со строкой `"est"`.
- Если требуется, чтобы символ присутствовал многократно или вообще отсутствовал в проверяемой на совпадение строке, после него следует указать знак звездочки (*****). Например, регулярное выражение `/t*est/` обеспечивает совпадение со строками `"test"`, `"ttest"`, `"tttest"` *и* `"est"`.
- Если требуется задать фиксированное число повторений символа в проверяемой на совпадение строке, это число следует указать в фигурных скобках после данного символа. Например, регулярное выражение `/a{4}/` означает проверку на совпадение с четырьмя подряд символами **a**.
- Если требуется задать число повторений символа в определенных пределах, эти пределы следует указать через запятую в фигурных скобках после проверяемого символа. Например, регулярное выражение `/a{4,10}/` обеспечивает совпадение с любой строкой, в которой следует подряд от четырех до десяти символов **a**.
- Если требуется задать число повторений символа в расширяемых пределах, эти пределы следует указать в фигурных скобках после проверяемого символа, опустив второе числовое значение, но оставив запятую. Например, регулярное выражение `/a{4,}/` обеспечивает совпадение с любой строкой, в которой следуют подряд четыре и больше символа **a**.

Любые из перечисленных выше операций повторения могут быть как *жадными*, так и *ленивыми*. По умолчанию эти операции жадные, т.е. они охватывают все символы, которые могут совпасть с сопоставляемым шаблоном. Если же указать операцию с последующим знаком вопроса (**?**), т.е. перегрузить операцию **?**, как, например, `a+?`, такая операция окажется ленивой. Это означает, что она будет охватывать лишь столько символов, сколько будет достаточно для совпадения.

Так, если требуется проверить совпадение с символьной строкой `"aaa"`, то регулярное выражение `/a+/` обеспечит совпадение со всеми тремя символами **a**, тогда как ленивое выражение `/a+?/` — только с одним символом **a**. Ведь для удовлетворения условия, задаваемого членом `a+` такого выражения, достаточно совпадения с единственным символом **a**.

Предопределенные классы символов

Имеются символы, которые требуется проверить на совпадение, но их нельзя указать буквально. К их числу относятся такие управляющие символы, как возврат каретки. Имеются также классы символов, которые часто проверяются на

совпадение. К их числу относятся наборы десятичных цифр или пробелов. Для представления подобных символов или общеупотребительных классов символов в синтаксисе регулярных выражений предусмотрен целый ряд предопределенных членов, с помощью которых можно проверить на соответствие управляющим символам. Это избавляет от необходимости определять свой набор символов при сопоставлении с общеупотребительными наборами символов в регулярных выражениях.

Все эти члены, а также отдельные управляющие символы или наборы символов, которые они представляют, перечислены в табл. 10.1. Эти предопределенные наборы помогают снять покров чрезмерной таинственности с регулярных выражений.

Таблица 10.1. Члены, обозначающие управляющие символы и предопределенные классы символов

Предопределенный член	Соответствует...
`\t`	горизонтальной табуляции
`\b`	символу забоя
`\v`	вертикальной табуляции
`\f`	прогону страницы
`\r`	возврату каретки
`\n`	переводу строки
`\cA` : `\cZ`	управляющим символам
`\u0000` : `\uFFFF`	шестнадцатеричным значениям символов в Юникоде
`\x00` : `\xFF`	шестнадцатеричным значениям символов в коде ASCII
`.`	любому символу, кроме перевода строки (`\n`)
`\d`	любой десятичной цифре, что эквивалентно выражению `[0-9]`
`\D`	любому символу, кроме десятичной цифры, что эквивалентно выражению `[^0-9]`
`\w`	любому буквенно-цифровому символу, включая и знак подчеркивания, что эквивалентно выражению `[A-Za-z0-9_]`
`\W`	любому символу, кроме буквенно-цифровых, включая и знак подчеркивания, что эквивалентно выражению `[^A-Za-z0-9_]`
`\s`	любому пробельному символу (собственно пробела, табуляции, перевода строки, прогона страницы и т.д.)
`\S`	любому не пробельному символу
`\b`	границе слова
`\B`	внутренней части слова, но не его границе

Группировка

В приведенных до сих пор примерах было показано, что операции, например `+` или `*`, воздействуют только на предшествующий им член регулярного выражения. Но если требуется применить операцию к группе членов, то для это-

го можно воспользоваться круглыми скобками, заключив в них группы членов регулярного выражения таким же образом, как это делается в любом математическом выражении. Например, регулярное выражение `/(ab)+/` обеспечивает совпадение с одним или более последовательными вхождениями подстроки `"ab"` в проверяемой строке.

Когда часть регулярного выражения заключается в круглые скобки как отдельная группа, она служит также для целей так называемой *фиксации*. Имеются самые разные виды фиксации, более подробно рассматриваемые в разделе 10.4.

Чередование (логическое ИЛИ)

Выбор альтернативных вариантов обозначается с помощью знака вертикальной черты (`|`). Например, регулярное выражение `/a|b/` обеспечивает совпадение с символом **a** или **b**, а регулярное выражение `/(ab)+|(cd)+/` — с одним или несколькими вхождениями подстроки `"ab"` или `"cd"`.

Обратные ссылки

К числу самых сложных членов, которые можно обозначить в регулярных выражениях, относятся обратные ссылки на *фиксации*, определяемые в этих выражениях. Подробнее о фиксациях речь пойдет в разделе 10.4, а до тех пор достаточно сказать, что их следует рассматривать как части проверяемой символьной строки, успешно совпавшие с членами регулярного выражения. Такие члены обозначаются обратной косой чертой и номером фиксации, на которую делается ссылка, начиная с **1**, например `\1`, `\2` и т.д.

В качестве примера рассмотрим регулярное выражение `/^([dtn])a\1/`, которое обеспечивает совпадение с символьной строкой, начинающейся с любого из символов **d**, **t** или **n**, за которыми следует символ **a** и далее любой символ, совпадающий с первой фиксацией. Последнее очень важно уяснить! Ведь это не одно и то же, что и выражение `/[dtn]a[dtn]/`. Символ, следующий после символа **a**, не может быть любым из символов **d**, **t** или **n**, но должен быть каким угодно из тех символов, которые инициируют совпадение с первым символом. Следовательно, символ, который совпадет с членом `\1`, неизвестен до самого момента вычисления данного регулярного выражения.

Характерным примером полезного применения обратных ссылок в регулярных выражениях может служить проверка на совпадение с элементами разметки XML-документов. Рассмотрим следующий пример:

```
/<(\w+)>(.+)<\/\1>/
```

С помощью такого регулярного выражения можно проверить на совпадение такие простые элементы разметки, как, например, `<strong>все что угодно</strong>`. Подобная проверка оказалась бы невозможной без указания обратной ссылки. Ведь заранее неизвестно, какой именно закрывающий дескриптор будет соответствовать открывающему.

А теперь, рассмотрев основные сведения о регулярных выражениях, перейдем к их благоразумному применению непосредственно в коде.

Совет

Приведенный выше материал можно сравнить с головокружительным ускоренным курсом по регулярным выражениям. Если они все еще кружат вам голову и вы чувствуете, что вам трудно будет усвоить последующий материал этой главы, настоятельно рекомендуем обратиться к дополнительной литературе, упоминавшейся в начале главы.

10.3. Компиляция регулярных выражений

Регулярные выражения проходят многочисленные стадии обработки, поэтому ясное представление о том, что происходит на каждой стадии этого процесса, способствует написанию оптимизированного кода на JavaScript. К наиболее примечательным среди них относятся стадии компиляции и выполнения. В частности, *компиляция* происходит всякий раз, когда регулярное выражение определяется впервые, а *выполнение* — когда скомпилированное регулярное выражение используется для сопоставления шаблонов с символьной строкой.

Во время компиляции интерпретатор JavaScript выполняет синтаксический анализ регулярного выражения и преобразует его во внутреннее представление, каким бы сложным оно ни было. Стадия синтаксического анализа и компиляции должна происходить всякий раз, когда в коде JavaScript встречается регулярное выражение (кроме любых внутренних оптимизаций, выполняемых браузером).

Зачастую браузеры способны *сами* определить, используются ли идентичные регулярные выражения, чтобы кешировать результаты компиляции конкретного повторяющегося выражения. Но это характерно далеко не для всех браузеров. Так, если речь идет о сложных выражениях, заметных улучшений в быстродействии можно добиться путем предварительного определения, а следовательно, и предварительной компиляции регулярного выражения для последующего его применения.

Как следует из приведенного выше краткого обзора регулярных выражений, в JavaScript они могут быть скомпилированы двумя способами: с помощью литерала или конструктора. Обратимся к конкретному примеру кода из листинга 10.2.

Листинг 10.2. Два способа создания скомпилированного регулярного выражения

```
const re1 = /test/i;                          <-- Создать регулярное выражение с помощью литерала
const re2 = new RegExp("test", "i");          <-- Создать регулярное выражение с помощью конструктора
assert(re1.toString()=== "/test/i",
       "Verify the contents of the expression.");
assert(re1.test("TesT"), "Yes, it's case-insensitive.");
assert(re2.test("TesT"), "This one is too.");
assert(re1.toString()=== re2.toString(),
       "The regular expressions are equal.");
assert(re1 !== re2, "But they are different objects.");
```

В данном примере кода оба регулярных выражения оказываются в конечном итоге в скомпилированном состоянии. Если заменить каждую ссылку на переменную `re1` литералом `/test/i`, то одно и то же регулярное выражение будет компилироваться снова и снова. Поэтому однократная компиляция регулярного выражения и последующее его сохранение в переменной может стать важной мерой для оптимизации кода.

Обратите внимание на то, что для каждого регулярного выражения создается уникальное объектное представление. Всякий раз, когда регулярное выражение определяется, а следовательно, и компилируется, для него создается также новый объект. Отличие регулярных выражений от других простых типов, включая строковые и числовые, в том и состоит, что их компиляция всегда дает уникальный результат.

Особое значение имеет применение конструктора `new RegExp(...)` для создания нового регулярного выражения. Такой способ позволяет составлять и компилировать выражение из символьной строки, динамически формируемой во время выполнения вашего приложения. И это может быть очень удобно для создания сложных выражений для их неоднократного использования.

Допустим, в документе требуется определить элементы с именем определенного класса, неизвестным до момента запуска приложения на выполнение. Имена элементов могут быть связаны с разными классами и храниться в неудобном формате, разделенными пробелами в символьной строке. Это представляет удобную возможность для компиляции регулярного выражения во время выполнения, как показано в примера кода из листинга 10.3.

Листинг 10.3. Компиляция регулярного выражения для последующего применения

```
<div class="samurai ninja"></div>
<div class="ninja samurai"></div>
<div></div>
<span class="samurai ninja ronin"></span>
<script>
  function findClassInElements(className, type) {
    const elems =
     document.getElementsByTagName(type || "*");
    const regex =
      new RegExp("(^|\\s)" + className + "(\\s|$)");
    const results = [];
    for (let i = 0, length = elems.length; i < length; i++) {
      if (regex.test(elems[i].className)) {
        results.push(elems[i]);
      }
    return results;
  }
  assert(findClassInElements("ninja", "div").length === 2,
         "The right amount of div ninjas was found.");
  assert(findClassInElements("ninja", "span").length === 1,
         "The right amount of span ninjas was found.");
  assert(findClassInElements("ninja").length === 3,
```

Создать объекты для тестирования, состоящие из различных элементов HTML-разметки с разными именами классов

Собрать элементы потипу

Скомпилировать регулярное выражение, используя переданное имя класса

Здесь будет сохраняться результат

Проверить регулярное выражение на совпадения

```
        "The right amount of ninjas was found.");
</script>
```

Из примера кода, приведенного в листинге 10.3, можно извлечь немало полезных уроков. Сначала в данном примере создается ряд элементов разметки `<div>` и `<span>` с именами классов в разных сочетаниях, которые мы будем использовать в качестве объектов для тестирования. Затем определяется функция для проверки имен классов, которой в качестве аргументов передаются имя проверяемого класса, а также тип элемента разметки, в котором он проверяется.

Далее собираются все элементы указанного типа с помощью встроенного метода `getElementsByTagName()` и составляется регулярное выражение, после чего оно компилируется следующим образом:

```
const regex = new RegExp("(^|\\s)" + className + "(\\s|$)");
```

Обратите внимание на применение конструктора `new RegExp()` для компилирования регулярного выражения на основании имени класса, передаваемого функции. В этом экземпляре объекта нельзя использовать литерал регулярного выражения, поскольку искомое имя класса заранее неизвестно.

Данное регулярное выражение составляется, а следовательно, и компилируется, только один раз, чтобы исключить частые и не нужные повторные компиляции. А поскольку содержимое регулярного выражения изменяется динамически (и зависит от входящего аргумента `className`), то, поступая подобным образом, можно добиться значительного выигрыша в производительности нашего приложения.

Само регулярное выражение обеспечивает совпадение с началом символьной строки или символом пробела, затем с целевым именем класса и далее с символом пробела или концом строки. Обратите особое внимание на использование двойной обратной косой черты (`\\`) для экранирования пробельного символа в выражении **`\\s`**. При создании регулярных выражений с помощью литералов обратная косая черта указывается в членах выражения только один раз. Но поскольку этот знак записывается в символьной строке, его необходимо экранировать, указывая дважды. Об этой особенности не следует забывать при составлении регулярных выражений в символьных строках, а не литералах.

Как только регулярное выражение будет скомпилировано, с его помощью можно будет собрать все совпадающие элементы, вызвав метод `test()` следующим образом:

```
regex.test(elems[i].className)
```

Предварительное составление и компиляция регулярных выражений с целью их повторного использования (и выполнения) настоятельно рекомендуется как эффективный способ повышения производительности программы, которым не стоит пренебрегать. Ведь такой способ приносит заметные выгоды практически во всех случаях применения сложных регулярных выражений.

В начале этого раздела упоминалось о том, что круглые скобки служат в регулярных выражениях не только для группировки их членов и выполнения над ними операций, но и для так называемых *фиксаций*. Теперь настало время рассмотреть это понятие более подробно, что и будет сделано в следующем разделе.

10.4. Фиксация совпадающих частей

Польза от применения регулярных выражений становится более очевидной, когда происходит *фиксация* полученных результатов для последующей их обработки тем или иным способом. Первым очевидным шагом на этом пути может стать простая проверка на совпадение символьной строки с шаблоном. И зачастую этого оказывается достаточно, хотя во многих случаях полезно также выяснить, *что* именно совпало в результате подобной проверки.

10.4.1. Выполнение простых фиксаций

Допустим, требуется извлечь значение из сложной символьной строки. Характерным примером такой строки может служить значение свойства CSS-преобразования, через которое можно видоизменить визуальное положение элемента разметки HTML-документа, как демонстрируется в примере кода из листинга 10.4.

Листинг 10.4. Простая функция для фиксации встраиваемого значения

```
<div id="square" style="transform:translateY(15px);"></div>
<script>
  function getTranslateY(elem){
    const transformValue = elem.style.transform;
    if(transformValue){
      const match = transformValue.match(/translateY\(([^\)]+)\)/);
      return match ? match[1] : "";
    }

    return "";
  }

  const square = document.getElementById("square");

  assert(getTranslateY(square) === "15px",
         "We've extracted the translateY value");
</script>
```

Определить объект для тестирования

Извлечь значение свойства `translateY` из символьной строки

Сначала в данном примере кода определяется элемент разметки, обозначающий стиль, изменяющий положение этого элемента на 15 пикселей:

```
"transform:translateY(15px);"
```

К сожалению, браузер не предоставляет интерфейс API, чтобы без особого труда извлечь величину сдвига элемента. Поэтому мы создаем для этой цели следующую функцию:

```
function getTranslateY(elem){
  const transformValue = elem.style.transform;
  if(transformValue){
    const match = transformValue.match(/translateY\(([^\)]+)\)/);
    return match ? match[1] : "";
  }
  return "";
}
```

Приведенный ниже код синтаксического анализа величины сдвига может показаться поначалу несколько запутанным, поэтому разберем его по частям.

```
const match = transformValue.match(/translateY\(([^\)]+)\)/);
return match ? match[1] : "";
```

Прежде всего необходимо определить, имеется ли вообще свойство `transform` для последующего синтаксического анализа его значения. Если оно отсутствует, возвращается пустая символьная строка. А если свойство `transform` присутствует, то можно перейти к извлечению значения этого свойства. Метод `match()` с регулярным выражением возвращает массив зафиксированных значений, если совпадение обнаружено, или же пустое значение `null`, если оно не обнаружено.

Массив, возвращаемый методом `match()`, содержит результат полного совпадения в первом своем элементе, а в каждом последующем элементе — остальные зафиксированные результаты. Таким образом, в первом элементе массива будет храниться полностью совпавшая символьная строка `"transform:translateY(15px);"`, а в следующем элементе — значение **`15px`**.

Напомним, что в регулярном выражении фиксации определяются с помощью круглых скобок. Следовательно, при соответствии значения свойства преобразования шаблону регулярного выражения, оно будет находиться во втором элементе массива (его индекс равен `[1]`), поскольку в данном регулярном выражении (после первой его части `translateY`) была указана в круглых скобках только одна фиксация.

В данном примере использованы локальное регулярное выражение и метод `match()`. Но совсем другое дело, когда используются глобальные регулярные выражения. Выясним далее, как это происходит.

10.4.2. Проверка на совпадение с помощью глобальных регулярных выражений

Как было показано в предыдущем разделе, если локальное регулярное выражение (без глобального флага **`g`**) используется в методе `match()`, вызываемом для объекта типа `String`, то из этого метода возвращается массив, содержащий

полностью совпавшую строку наряду с любыми другими результатами совпадений, зафиксированными в ходе данной операции.

Но если предоставить методу `match()` глобальное регулярное выражение (с глобальным флагом **g**), то будет возвращен совсем другой результат. И хотя им по-прежнему окажется массив, он будет содержать результаты глобальных совпадений. Ведь глобальное регулярное выражение обеспечивает все возможные варианты совпадений в проверяемой символьной строке, а не только первое совпадение. И в этом случае результаты, зафиксированные *при каждом* совпадении, не возвращаются.

Покажем, как это происходит, на примере кода из листинге 10.5.

Листинг 10.5. Отличия глобального и локального поиска на совпадение с помощью метода `match`

```
const html = "<div class='test'><b>Hello</b> <i>world!</i></div>";
const results = html.match(/<(\/?)(\w+)([^>]*?)>/);
assert(results[0] === "<div class='test'>", "The entire match.");
assert(results[1] === "", "The (missing) slash.");
assert(results[2] === "div", "The tag name.");
assert(results[3] === " class='test'", "The attributes.");

const all = html.match(/<(\/?)(\w+)([^>]*?)>/g);
assert(all[0] === "<div class='test'>", "Opening div tag.");
assert(all[1] === "<b>", "Opening b tag.");
assert(all[2] === "</b>", "Closing b tag.");
assert(all[3] === "<i>", "Opening i tag.");
assert(all[4] === "</i>", "Closing i tag.");
assert(all[5] === "</div>", "Closing div tag.");
```

Проверить на совпадение с помощью локального регулярного выражения

Проверить на совпадение с помощью глобального регулярного выражения

Как следует из приведенного выше примера кода, при локальном совпадении регулярного выражения `html.match(/<(\/?)(\w+)([^>]*?)>/)`, возвращается единственный совпавший экземпляр и зафиксированные при этом результаты, а при глобальном совпадении, `html.match(/<(\/?)(\w+)([^>]*?)>/g)` — список совпадений. Если же особое значение имеют фиксации, их можно восстановить, выполняя глобальный поиск совпадений и вызывая для этой цели метод `exec()` с регулярным выражением всякий раз, когда требуется возвратить очередную порцию совпавших данных. Типичный тому пример приведен в коде из листинга 10.6.

Листинг 10.6. Глобальный поиск и фиксация результатов с помощью метода `exec()`

```
const html = "<div class='test'><b>Hello</b> <i>world!</i></div>";
const tag = /<(\/?)(\w+)([^>]*?)>/g;
let match, num = 0;
while ((match = tag.exec(html)) !== null) {
  assert(match.length === 4,
         "Every match finds each tag and 3 captures.");
  num++;
}
assert(num === 6, "3 opening and 3 closing tags found.");
```

Многократно вызывать метод `exec()`

В приведенном выше примере кода метод `exec()` вызывается многократно, как показано ниже.

```
while ((match = tag.exec(html)) !== null) {...}
```

Благодаря этому сохраняется состояние из предыдущего вызова. Таким образом, при каждом последующем вызове данного метода происходит переход к следующему глобальному совпадению. После каждого вызова метода `exec()` возвращается очередное совпадение *и* его фиксации.

С помощью методов `match()` или `exec()` можно всегда обнаружить точные совпадения (и фиксации) искомых результатов. Но этого оказывается недостаточно, когда возникает потребность в обратной ссылке на сами фиксации.

10.4.3. Ссылки на фиксации

Ссылаться на отдельные части зафиксированных результатов совпадения можно двумя способами: в совпадении или в строке замены (там, где это, конечно, возможно). Вернемся к примеру из листинга 10.6, где обнаруживается совпадение с открывающим или закрывающим дескриптором HTML-разметки. Видоизменим его таким образом, чтобы обнаруживалось также совпадение с внутренним содержимым дескриптора, как показано в примере кода из листинга 10.7.

Листинг 10.7. Применение обратных ссылок для проверки на совпадение с содержимым дескриптора HTML-разметки

```
const html = "<b class='hello'>Hello</b> <i>world!</i>";
const pattern = /<(\w+)([^>]*)>(.*?)<\/\1>/g;   <-- Использовать обратную ссылку
let match = pattern.exec(html);                 <-- Сопоставить шаблон с проверяемой строкой
assert(match[0] === "<b class='hello'>Hello</b>",   <-- Проверить различные фиксации по определенному шаблону
       "The entire tag, start to finish.");
assert(match[1] === "b", "The tag name.");
assert(match[2] === " class='hello'", "The tag attributes.");
assert(match[3] === "Hello", "The contents of the tag.");

match = pattern.exec(html);
assert(match[0] === "<i>world!</i>",
       "The entire tag, start to finish.");
assert(match[1] === "i", "The tag name.");
assert(match[2] === "", "The tag attributes.");
assert(match[3] === "world!", "The contents of the tag.");
```

В приведенном выше примере кода член `\1` используется в регулярном выражении для обратной ссылки на первую фиксацию в этом выражении (в данном случае это имя дескриптора). Используя эту информацию, можно обнаружить совпадение с соответствующим закрывающим дескриптором по обратной ссылке на исходный дескриптор, но при условии, что в текущем дескрипторе отсутствуют другие дескрипторы с таким же именем. Поэтому данный пример едва ли можно считать исчерпывающим.

Ссылки на фиксации в заменяющей строке можно получить и с помощью метода `replace()`. Вместо кодов обратной ссылки, как в предыдущем примере кода, в данном случае используется синтаксис `$1`, `$2`, `$3` и так далее для ссылки на каждую фиксацию по ее номеру:

```
assert("fontFamily".replace(/([A-Z])/g, "-$1").toLowerCase() ==
      "font-family", "Convert the camelCase into dashed notation.");
```

В этом фрагменте кода ссылка (`$1`) на первое зафиксированное значение (в данном случае прописную букву **F**) делается в *строке замены*. Это позволяет указывать строку замены, даже не зная, каким будет ее значение до самого момента совпадения. Такой эффективный прием стоит взять на вооружение!

Наличие ссылок на фиксации в регулярных выражениях способствует существенному упрощению и удобочитаемости кода. Выразительный характер таких ссылок позволяет сделать краткими и ясными операторы, которые в противном случае получились бы довольно запутанными, неясными и длинными.

Поскольку и фиксации и группы выражений заключаются в круглые скобки, процессору регулярных выражений неизвестно, какие именно скобки используются для группировки членов регулярного выражения и какие из них служат для обозначения фиксаций. Поэтому все они интерпретируются и как группы и как фиксации, что может привести к фиксации большего объема информации, чем требуется на самом деле. Как же поступать в подобных случаях? Ответ на этот вопрос дается в следующем разделе.

10.4.4. Нефиксируемые группы

Как отмечалось ранее, круглые скобки имеют двойное назначение: они не только группируют члены регулярного выражения для выполнения операций, но и обозначают фиксации. Обычно это не вызывает особых затруднений, но если в регулярном выражении происходит частая группировка, то в конечном итоге может быть зафиксировано много лишней информации. А это, в свою очередь, затрудняет сортировку зафиксированных результатов. В качестве примера рассмотрим следующее регулярное выражение:

```
const pattern = /((ninja-)+)sword/;
```

В данном примере преследуется цель составить такое регулярное выражение, которое должно распознавать префикс `"ninja-"` один или больше раз перед словом `"sword"`, а также к фиксации всего префикса. Для этого в данном регулярном выражении требуются два ряда круглых скобок.

- Внешние скобки обозначают фиксацию всего, что предшествует символьной строке `"sword"`.
- Внутренние скобки группируют префикс `"ninja-"` для применения операции **+**.

Все это верно, но в результате возникает не одна предполагаемая фиксация, а больше из-за наличия внутренних группирующих скобок. Для указания на то,

что круглые скобки не должны приводить к фиксации, в синтаксисе регулярных выражений предусмотрено обозначение `?:`, которое делается сразу же после открывающей круглой скобки. Это обозначение называется *пассивным подвыражением*.

Следовательно, если заменить упомянутое выше регулярное выражение на следующее:

```
const pattern = /((?:ninja-)+)sword/;
```

то к фиксации результатов приведут только внешние круглые скобки. А внутренние круглые скобки в этом выражении будут преобразованы в пассивное подвыражение. Чтобы убедиться в этом на практике, рассмотрим пример кода из листинга 10.8.

Листинг 10.8. Группировка без фиксации

```
const pattern = /((?:ninja-)+)sword/;      <-- Использовать пассивное подвыражение
const ninjas = "ninja-ninja-sword".match(pattern);

assert(ninjas.length === 2,"Only one capture was returned.");
assert(ninjas[1] === "ninja-ninja-",
       "Matched both words, without any extra capture.");
```

Результаты выполнения тестов в приведенном выше примере кода показывают, что пассивное подвыражение `/((?:ninja-)+)sword/` препятствует появлению лишних фиксаций. Поэтому при всякой возможности следует стремиться использовать нефиксируемые (т.е. пассивные) группы вместо фиксации, когда она не требуется, чтобы облегчить работу анализатору регулярных выражений по запоминанию и возврату фиксаций. Ведь если зафиксированные результаты не нужны, то зачем их запрашивать! Правда, за это приходится платить усложнением и без того запутанных регулярных выражений.

А теперь обратим внимание на еще один способ раскрытия истинного потенциала регулярных выражений. И состоит он в применении функций вместе с методом `replace()` объекта типа `String`.

10.5. Замена текста с помощью функций

Метод `replace()` объекта типа `String` довольно эффективен и универсален, в чем мы уже успели убедиться ранее при обсуждении фиксаций. Если в качестве первого аргумента вместо искомой строки передать этому методу регулярное выражение, то в исходной строке будет заменена подстрока, совпадающая с шаблоном (или *все* подстроки, если регулярное выражение является глобальным).

Допустим, что в символьной строке требуется заменить все символы верхнего регистра на букву `"X"`. Для этого можно было бы составить приведенное ниже регулярное выражение, которое дало бы в результате символьную строку `"XXXXXfg"`.

```
"ABCDEfg".replace(/[A-Z]/g,"X")
```

Это, конечно, замечательно, но, вероятно, наиболее сильной стороной метода `replace()` является возможность предоставить в качестве заменяющего значения функцию, а не фиксированную символьную строку. Когда заменяющее значение (второй аргумент данного метода) оказывается функцией, она вызывается для каждого обнаруженного совпадения с переменным списком параметров приводимого ниже содержания (напомним, что при глобальном поиске в исходной символьной строке обнаруживаются все экземпляры совпадения с шаблоном). А значение, возвращаемое данной функцией, служит в качестве заменяющего.

- Полный текст совпадения.
- Фиксации совпадений по каждому параметру.
- Индекс совпадения в исходной символьной строке.
- Исходная символьная строка.

Это дает немалую свободу действий для определения содержимого строки замены во время выполнения программы, причем большая часть этой информации зависит от характера искомого совпадения. В листинге 10.9 приведен пример кода, в котором функция динамически формирует заменяющие значения в процессе преобразования символьной строки со словами, разделенными тире, в эквивалентное им смешанное написание.

Листинг 10.9. Преобразование написанной через тире символьной строки в смешанное написание

```
function upper(all,letter) { return letter.toUpperCase(); }
assert("border-bottom-width".replace(/-(\w)/g,upper)
       === "borderBottomWidth",
       "Camel cased a hyphenated string.");
```

Преобразовать в верхний регистр

Сопоставить с символами, написанными через тире

В приведенном выше примере кода предоставляется глобальное регулярное выражение, обеспечивающее совпадение с любым символом, которому предшествует знак тире. А фиксация в этом выражении обозначает совпавший символ (без тире). Всякий раз, когда вызывается функция `upper()`, а в данном случае это происходит дважды, ей передается полностью совпавшая символьная строка в качестве первого аргумента, а также фиксация (только один раз для этого регулярного выражения) в качестве второго аргумента. Остальные аргументы не важны, поэтому они и не указываются.

При первом вызове функции `upper()` ей передаются аргументы `"-b"` и `"b"`, а при втором вызове — аргументы `"-w"` и `"w"`. В каждом случае зафиксированная строчная буква преобразуется в прописную и возвращается в качестве заменяющей символьной строки. И в конечном итоге подстрока `"-b"` заменяется на **`"B"`**, а подстрока `"-w"` — на `"W"`.

Глобальное регулярное выражение обусловливает выполнение данной функции замены всякий раз, когда происходит совпадение в исходной символьной строке. Поэтому рассмотренный здесь способ можно даже расширить за пределы обычной замены, чтобы использовать его в качестве средства для обхода символьных строк и своего рода альтернативы типичному применению метода `exec()` в цикле `while`, как было показано ранее в этой главе.

Допустим, требуется преобразовать следующую строку запроса:

```
"foo=1&foo=2&blah=a&blah=b&foo=3"
```

в приведенный ниже альтернативный формат, в большей степени подходящий преследуемым целям.

```
"foo=1,2,3&blah=a,b"
```

Для решения этой задачи можно воспользоваться регулярным выражением и методом `replace()` в коде, который получается особенно кратким, как демонстрируется в примере кода из листинга 10.10.

Листинг 10.10. Способ сжатия строки запроса

```
function compress(source) {
  const keys = {};                    <-- Сохранить расположенные ключи
  source.replace(
    /([^=&]+)=([^&]*)/g,
    function(full, key, value) {      <-- Извлечь данные о ключе и значении
      keys[key] =
        (keys[key] ? keys[key] + "," : "") + value;
      return "";
    }
  );
  const result = [];
  for (let key in keys) {
    result.push(key + "=" + keys[key]);   <-- Собрать данные о ключе
  }
  return result.join("&");            <-- Соединить результаты знаком &
}
assert(compress("foo=1&foo=2&blah=a&blah=b&foo=3") ===
               "foo=1,2,3&blah=a,b",
               "Compression is OK!");
```

Самое любопытное в коде из листинга 10.10 состоит в том, что метод `replace()` замены символьной строки служит скорее в качестве средства обхода этой строки для обнаружения отдельных значений, а не механизма поиска и замены. Такой прием преследует две цели: передать функцию в качестве аргумента заменяющего значения при вызове метода `replace()` и просто использовать ее для поиска вместо возврата значения.

Сначала в рассматриваемом здесь примере кода объявляется информационный массив `keys` для хранения ключей и значений, обнаруживаемых в исходной строке запроса. Затем для исходной строки вызывается метод `replace()`, которому передается регулярное выражение для поиска на совпадение пар

"ключ–значение" и фиксации ключа и значения, а также функция, которой в качестве аргументов передаются результат полного совпадения, зафиксированный ключ и зафиксированное значение. Эти зафиксированные величины сохраняются в хеш-массиве для последующего обращения к ним. Следует заметить, что из этой функции возвращается пустая символьная строка, поскольку в данном случае используются побочные эффекты, а не сам результат замены в исходной строке, который особого значения не имеет.

После возврата из метода `replace()` объявляется обычный массив, в котором будут накапливаться результаты, и далее организуется циклическое обращение к обнаруженным ключам, в ходе которого каждый ключ добавляется в данный массив. И наконец, отдельные результаты, хранящиеся в массиве, соединяются разграничительным знаком **&** и возвращается полученный результат:

```
const result = [];
for (let key in keys) {
  result.push(key + "=" + keys[key]);
}
return result.join("&");
```

Подобным способом метод `replace()` объекта `String` можно использовать в качестве самостоятельного механизма поиска в символьных строках, добиваясь результата не только быстро, но и просто и эффективно. Потенциальные возможности такого способа трудно переоценить, особенно если учесть, что для этого требуется совсем немного кода.

На самом деле регулярные выражения могут оказывать заметное влияние и на способ написания сценариев для веб-страниц. Покажем теперь, как применить знания, полученные о регулярных выражениях, для решения типичных задач, которые могут возникнуть при разработке веб-приложений на JavaScript.

10.6. Решение типичных задач с помощью регулярных выражений

В языке JavaScript часто одни и те же задачи повторяются снова и снова, но их решение не всегда очевидно. И здесь на помощь может прийти мастерское владение регулярными выражениями. В этом разделе будут рассмотрены некоторые типичные задачи, которые можно решить с помощью одного или двух регулярных выражений.

10.6.1. Учет символов перехода на новую строку

При выполнении поиска нередко требуется, чтобы член . (точка), который обычно совпадает с любым символом, кроме символа перевода строки, охватывал бы и символы новой строки. Для этой цели в других языках программирования при составлении регулярных выражений зачастую используется специальный флаг, а в JavaScript этого сделать нельзя. Поэтому рассмотрим два

способа возмещения этого явного упущения в JavaScript, как демонстрируется в примере кода из листинга 10.11.

Листинг 10.11. Анализ *всех* символов строки с учетом символа перехода на новую строку

```
                                    Определить объект для тестирования
const html = "<b>Hello</b>\n<i>world!</i>";
assert(/.*/.exec(html)[0] === "<b>Hello</b>",
       "A normal capture doesn't handle endlines.");
assert(/[\S\s]*/.exec(html)[0] ===
       "<b>Hello</b>\n<i>world!</i>",
       "Matching everything with a character set.");
assert(/(?:.|\s)*/.exec(html)[0] ===
       "<b>Hello</b>\n<i>world!</i>",
       "Using a non-capturing group to match everything.");
```

Показать, что совпадения с символами перехода на новую строку не произошло

Использовать сопоставление с пробельными символами для совпадения со всеми символами

Использовать чередование для совпадения со всеми символами

Сначала в данном примере кода определяется символьная строка `"<b>Hello</b>\n<i>world!</i>"`, служащая в качестве объекта тестирования и содержащая знак перевода строки. Затем предпринимается попытка разными способами сопоставить шаблон регулярного выражения всем символам строки.

В первом тесте, `/.*/.exec(html)[0] === "<b>Hello</b>"`, мы убеждаемся, что оператор точки (`.`) не охватывает символ перевода строки. Но настоящие мастера своего дела не отступают перед трудностями, поэтому в следующем тесте для проверки используется альтернативный вариант регулярного выражения, `/[\S\s]*/`, в котором определяется класс символов для совпадения с любыми символами, которые *не* являются пробельными, а также с любыми символами, которые *относятся* к пробельным. Такое сочетание охватывает весь набор символов, в том числе и перехода на новую строку.

Еще одна попытка предпринимается в следующем тесте:

```
/(?:.|\s)*/.exec(html)[0] === "<b>Hello</b>\n<i>world!</i>"
```

Здесь используется регулярное выражение `/(?:.|\s)*/`, обеспечивающее с помощью оператора точки (`.`) совпадение со всеми символами, кроме перевода строки, а также с любыми пробельными символами, включая и перевод строки. Такое сочетание охватывает весь набор символов, в том числе и знак перевода строки. Обратите внимание на использование в данном случае пассивного подвыражения, исключающего любые нежелательные фиксации.

Очевидно, что самым оптимальным является решение с помощью регулярного выражения `/[\S\s]*/` благодаря его простоте и подразумеваемым преимуществам в быстродействии.

А теперь сделаем смелый шаг, чтобы расширить кругозор до мирового горизонта.

10.6.2. Сопоставление с символами Юникода

Нередко в регулярном выражении требуется сопоставить с шаблоном набор буквенно-цифровых символов, например для выделения идентификатора из селектора CSS. Но было бы недальновидно предполагать, что все буквенно-цифровые символы соответствуют стандарту ASCII и включают только буквы английского алфавита.

В этой связи целесообразно расширить набор используемых символов до стандарта Юникод, чтобы явным образом можно было поддерживать многие языки мира. Тогда у вас появится возможность использовать набор буквенно-цифровых символов, который не входит в стандарт ASCII, как демонстрируется в примере кода из листинга 10.12.

Листинг 10.12. Сопоставление с символами Юникода

```
const text ="\u5FCD\u8005\u30D1\u30EF\u30FC";
const matchAll = /[\w\u0080-\uFFFF_-]+/;
assert(text.match(matchAll),"Our regexp matches non-ASCII!");
```

Сопоставить со всеми символами, в том числе и с заданными в стандарте Юникод

В приведенном выше примере кода проверка на совпадение охватывает все символы Юникода, для чего создается класс символов, включающий член `\w` для сопоставления со всеми обычными символами с кодом ASCII до **`128`** в десятичной системе и **`0x80`** в шестнадцатеричной системе счисления, а также диапазон, охватывающий весь набор символов Юникода с десятичным кодом свыше **`128`** (или **`U+0x80`** в шестнадцатеричной форме). Начиная с десятичного кода **`128`**, обнаруживаются некоторые символы из верхней половины набора в коде ASCII и все символы в основной многоязыковой плоскости Юникода.

Проницательные читатели могли заметить, что введением в регулярное выражение всего диапазона символов Юникода свыше кода **`\u0080`** в данном примере обеспечивается совпадение не только с буквенными символами, но и со всеми знаками препинания и другими специальными символами Юникода (например, стрелками). В этом нет никакой погрешности, поскольку основное назначение данного примера — показать, как вообще осуществляется проверка на совпадение с символами Юникода. Если же у вас имеется какой-то определенный диапазон символов для сопоставления, вы можете воспользоваться данным примером, чтобы ввести любой нужный вам диапазон в класс символов.

И в заключение нашего исследования потенциальных возможностей регулярных выражений рассмотрим еще одну весьма распространенную задачу.

10.6.3. Сопоставление с экранированными символами

Разработчики веб-страниц часто пользуются именами идентификаторов элементов разметки страницы (`id`), совпадающими с идентификаторами, используемыми в программе. Однако здесь все зависит от принятых соглашений. Но ведь значения `id` могут содержать не только обычные символы, но и знаки

препинания. В частности, разработчик может присвоить элементу разметки значение id, равное form:update.

Реализуя, например, механизм CSS-селекторов, разработчик библиотеки может организовать поддержку знаков препинания, применяя экранирование символов. Это дает пользователю возможность указывать сложные имена, которые не соответствуют принятым соглашениям о присвоении имен. Рассмотрим в качестве примера код из листинга 10.13.

Листинг 10.13. Сопоставление с экранированными символами в CSS-селекторе

```
const pattern = /^((\w+)|(\\.))+$/;
const tests = [
    "formUpdate",
    "form\\.update\\.whatever",
    "form\\:update",
    "\\f\\o\\r\\m\\u\\p\\d\\a\\t\\e",
    "form:update"
];

for (let n = 0; n < tests.length; n++) {
  assert(pattern.test(tests[n]),
         tests[n] + " is a valid identifier" );
}
```

Создать различные тестируемые объекты. Все они должны пройти тест, кроме последнего, где специальный знак (:) остался не экранированым

Это регулярное выражение обеспечивает совпадение с любой последовательностью, состоящей из буквенных символов, обратной косой черты и следующего за ней любого символа (даже еще одного символа обратной косой черты) или того и другого

Проверить все тестируемые объекты

В данном конкретном выражении обнаруживается совпадение с последовательностью буквенных символов или последовательностью символов обратной косой черты, за которыми следует любой символ. Следует, однако, иметь в виду, что для полной поддержки всех экранированных символов придется приложить дополнительные усилия. Подробнее об этом можно узнать, обратившись по адресу https://mathiasbynens.be/notes/css-escapes.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Регулярные выражения являются весьма эффективным средством, все чаще проникающим буквально во все аспекты разработки современных веб-приложений на JavaScript, обеспечивая любую проверку на совпадение в зависимости от их конкретного применения. Хорошо разбираясь в основных принципах составления и применения регулярных выражений, рассмотренных в этой главе, любой разработчик должен уверенно чувствовать себя, преодолевая трудности при написании тех фрагментов кода, где можно выгодно воспользоваться регулярными выражениями.
- Регулярные выражения можно составлять с помощью литералов регулярных выражений (/test/) и конструктора объектов типа RegExp (new RegExp("test")). Литералы более предпочтительны, когда регулярное

выражение известно на стадии разработки, тогда как конструктор — когда регулярное выражение составляется во время выполнения программы.

- С регулярным выражением можно связать пять флагов. В частности, флаг **i** делает регулярное выражение независящим от регистра символов, флаг **g** обеспечивает сопоставление со всеми экземплярами шаблона, флаг **m** позволяет применить регулярное выражение к нескольким строкам текста, флаг **y** активизирует "липкое" сопоставление, тогда как флаг **u** позволяет пользоваться управляющими последовательностями в стандарте Юникод. Все эти флаги вводятся в конце литерала регулярного выражения (например, `/test/ig`) или в качестве второго аргумента при вызове конструктора объектов типа `RegExp` (например, `new RegExp("test", "i")`).
- Для указания набора сопоставляемых символов используется обозначение **[]** (например, `[abc]`).
- Знак вставки (**^**) служит для указания на то, что шаблон должен находиться в начале символьной строки, а знак денежной единицы (**$**) — на то, что шаблон должен находиться в конце строки.
- Знак **?** служит для указания на то, что член является необязательным, знак **+** — на то, что член должен появляться в регулярном выражении один или несколько раз, а знак ***** — на то, что член появляется один раз, несколько раз или вообще отсутствует в регулярном выражении.
- Знак точки (**.**) служит для сопоставления с любым символом.
- Знак обратной косой черты (****) служит для экранирования служебных символов в регулярном выражении (например, `. [ $ ^`).
- Круглые скобки **()** служат для группировки нескольких членов регулярного выражения, а знак конвейера (**|**) — для обозначения чередования, т.е. выбора альтернативных вариантов.
- На части символьной строки, успешно совпавшие с членами регулярного выражения, можно делать обратные ссылки, указывая соответствующее число, обозначающее номер фиксации, после знака обратной черты (например, `\1`, `\2` и т.д.).
- Для всякой символьной строки доступен метод `match()`, которому передается в качестве параметра регулярное выражение, а он возвращает массив, содержащий всю совпавшую строку наряду с любыми совпавшими фиксациями. А для замены совпадений по шаблону, а не части фиксированной строки, служит метод `replace()`.

Упражнения

1. Какими из приведенных ниже языковых средств могут быть составлены регулярные выражения в JavaScript?

а) Литералы регулярных выражений.

б) Встроенный конструктор объектов типа `RegExp`.

в) Встроенный конструктор объектов типа `RegularExpression`.

2. Что из приведенного ниже является литералом регулярного выражения?

а) `/test/`

б) `\test\`

в) `new RegExp("test");`

3. В каком из приведенных ниже регулярных выражений правильно указаны флаги?

а) `test/g`

б) `g/test/`

в) `new RegExp("test", "gi");`

4. С какой из приведенных ниже символьных строк совпадает регулярное выражение `/def/`?

а) Одна из символьных строк `"d"`, `"e"`, `"f"`

б) `"def"`

в) `"de"`

5. С какой из приведенных ниже символьных строк совпадает регулярное выражение `/[^abc]/`?

а) Одна из символьных строк `"a"`, `"b"`, `"c"`

б) Одна из символьных строк `"d"`, `"e"`, `"f"`

в) Символьная строка `"ab"`

6. С каким из приведенных ниже регулярных выражений совпадает символьная строка `"hello"`?

а) `/hello/`

б) `/hell?o/`

в) `/hel*o/`

г) `/[hello]/`

7. С какой из приведенных ниже символьных строк совпадает регулярное выражение `/(cd)+(de)*/`?

а) `"cd"`

б) `"de"`

в) `"cdde"`

г) `"cdcd"`

д) `"ce"`

е) `"cdcddededе"`

8. Каким из приведенных ниже знаков можно обозначить альтернативные варианты в регулярном выражении?

   а) **#**

   б) **&**

   в) **|**

9. Каким из приведенных ниже способов можно сослаться на первую совпавшую цифру в регулярном выражении `/([0-9])2/`?

   а) `/0`

   б) `/1`

   в) `\0`

   г) `\1`

10. С каким из приведенных ниже чисел совпадет регулярное выражение `/([0-5])6\1/`?

    а) **060**

    б) **16**

    в) **261**

    г) **565**

11. С какой из приведенных ниже символьных строк совпадет регулярное выражение `/(?:ninja)-(trick)?-\1/`?

    а) `"ninja-"`

    б) `"ninja-trick-ninja"`

    в) `"ninja-trick-trick"`

12. К какому из приведенных ниже результатов приведет вызов `"012675".replace(/[0-5]/g, "a")`?

    а) `"aaa67a"`

    б) `"a12675"`

    в) `"a1267a"`

11 Методики модуляризации кода

В этой главе...

- Применение проектного шаблона Модуль
- Написание модульного кода согласно текущим стандартам AMD и CommonJS
- Работа с модулями в стандарте ES6

До сих пор мы исследовали такие основные примитивы JavaScript, как функции, объекты, коллекции и регулярные выражения. Но в арсенале средств разработчика имеются и другие инструментальные средства для решения текущих задач написания прикладного кода на JavaScript. По мере расширения веб-приложений возникает ряд других задач, связанных со структурированием прикладного кода и его управлением. Как было неоднократно доказано, крупные монолитные кодовые базы намного труднее понять и сопровождать, чем более мелкие и хорошо организованные. Поэтому вполне естественно, что единственный способ усовершенствовать структуру и организацию прикладных программ состоит в том, чтобы разделить их на более мелкие, относительно слабо связанные части, называемые *модулями*.

Модули являются более крупными единицами организации прикладного кода, чем объекты и функции. Они позволяют разделять прикладные программы на связанные вместе блоки. При создании модулей следует стремиться образовать согласованные абстракции и инкапсулировать подробности реализации. Благодаря этому упрощается осмысление прикладной программы, поскольку, используя функциональные возможности модулей, можно не беспокоиться о малозначащих подробностях. Кроме того, наличие модулей означает возмож-

ность без особого труда повторно использовать функциональные средства в разных частях одной прикладной программы и даже разных прикладных программ, значительно ускоряя процесс разработки.

Как пояснялось ранее, в JavaScript повсеместно применяются глобальные переменные. Всякий раз, когда переменная определяется в главном коде, она автоматически становится глобальной и может быть доступна в любой другой части прикладного кода. В небольших программах это обычно не вызывает особых затруднений, но по мере расширения прикладных программ и включения в них стороннего кода заметно увеличивается вероятность конфликта имен. В большинстве других языков программирования подобное затруднение разрешается с помощью пространств имен (в C++ и C#) или пакетов (в Java). Они позволяют изолировать все используемые в них имена и назначить им другое имя. Благодаря этому значительно снижается вероятность появления конфликта имен.

До появления стандарта ES6 в языке JavaScript отсутствовало высокоуровневое встроенное средство, позволяющее группировать связанные вместе переменные в модуль, пространство имен или пакет. Поэтому в качестве выхода из подобного затруднительного положения программирующие на JavaScript выработали усовершенствованные методики модуляризации, в которых выгодно используются преимущества существующих языковых конструкций JavaScript, в том числе объектов, немедленно вызываемых функций и замыканий. Такие методики будут исследованы в этой главе.

Правда, только время покажет, как долго придется пользоваться подобными обходными приемами, поскольку в стандарт ES6 наконец-то были внедрены собственные модули. Но, к сожалению, браузеры не поспевают за этими нововведениями, и поэтому мы выясним, каким образом модули должны действовать в стандарте ES6, несмотря на то, что проверить их пока еще нельзя из-за отсутствия конкретной реализации в отдельных браузерах.

Знаете ли вы?

Каким из имеющихся механизмов можно воспользоваться для приближенного представления модулей в коде JavaScript до появления стандарта ES6?

Чем отличаются стандарты определения модулей AMD и CommonJS?

Какие операторы потребуются в коде, начиная со стандарта ES6, чтобы вызвать в модуле `guineaPig` функцию `tryThisOut()` из модуля `test`?

Итак, начнем с рассмотрения методик модуляризации, которыми можно пользоваться в настоящее время.

11.1. Модуляризация кода JavaScript до появления стандарта ES6

До появления стандарта ES6 в языке JavaScript допускались лишь следующие две области видимости: глобальная и локальная (область видимости функции).

В языке не существовало промежуточной области видимости (например, пространства имен или модуля), которая бы позволяла сгруппировать определенные функциональные возможности. Чтобы написать модульный код, разработчики веб-приложений на JavaScript были вынуждены творчески подходить к применению имеющихся языковых средств JavaScript.

Выбирая подходящие языковые средства, необходимо иметь в виду, что каждая модульная система должна быть способна как минимум на следующее.

- *Определить интерфейс*, через который можно получить доступ к функциональным возможностям, которые предоставляет модуль.
- *Скрыть внутренние подробности реализации модуля*, чтобы не обременять ими излишне пользователей модуля. Скрывая внутренние подробности реализации модуля, можно также защитить их от вмешательства извне, предотвратив тем самым ненужные модификации, способные породить всевозможные побочные эффекты и программные ошибки.

В этом разделе мы сначала покажем, как создавать модули, пользуясь стандартными языковыми средствами JavaScript, рассмотренными ранее в данной книге, в том числе объектами, замыканиями и немедленно вызываемыми функциями. Продолжая рассмотрение особенностей модуляризации, мы исследуем два наиболее распространенных стандарта определения модулей в JavaScript: AMD (Asynchronous Module Definition – асинхронное определение модуля) и CommonJS, которые построены на немного разных принципах. По ходу изложения материала этого раздела вы узнаете, как определять модули по этим стандартам и каковы их достоинства и недостатки. Но начнем мы с того, для чего была подготовлена почва в предыдущих главах.

11.1.1. Определение модулей с помощью объектов, замыканий и немедленно вызываемых функций

Итак, вернемся к перечисленным выше минимальным требованиям к модульной системе, которые состоят в том, чтобы скрыть внутренние подробности реализации и определить интерфейсы модулей. И теперь рассмотрим те языковые средства, которыми можно выгодно воспользоваться для реализации этих требований.

- **Сокрытие внутренних подробностей реализации модулей.** Как известно, при вызове функции в JavaScript образуется новая область видимости, в которой можно определить переменные, доступные только в теле данной функции. Следовательно, одна из возможностей скрыть внутренние подробности реализации модуля заключается в том, чтобы воспользоваться функциями в качестве модулей. Благодаря этому все переменные функции становятся внутренними переменными модуля, скрытыми от внешнего мира.

- **Определение интерфейсов модулей.** Реализация внутренних подробностей функционирования модулей через переменные функций означает, что эти переменные доступны только в самом модуле. Но если модули предполагается использовать в другом коде, то придется определить ясный интерфейс, через который можно получить доступ к функциональным возможностям, предоставляемым модулем. Чтобы добиться этого, можно, например, воспользоваться преимуществами объектов и замыканий. Идея состоит в том, чтобы из функции модуля возвращался объект, представляющий открытый интерфейс этого модуля. У этого объекта должны быть методы, которые через замыкания сохраняют активными внутренние переменные модуля — даже после того, как функция модуля завершит свое выполнение.

Описав в общих чертах, каким образом модули реализуются в JavaScript, перейдем к постепенному рассмотрению способов реализации требований к модулям, начав с применения функций для сокрытия внутренних подробностей реализации модулей.

Функции в качестве модулей

При вызове функции образуется новая область видимости, в которой определяются переменные, недоступные за пределами текущей функции. В качестве примера рассмотрим следующий фрагмент кода, в котором подсчитывается количество щелчков левой кнопкой мыши на веб-странице:

```
(function countClicks(){
  let numClicks = 0;
  document.addEventListener("click", () => {
    alert( ++numClicks );
  });
})();
```

Определить локальную переменную для хранения подсчета количества щелчков

Всякий раз, когда пользователь щелкает кнопкой мыши на веб-странице, инкрементируется счетчик и сообщается его текущее значение

В приведенном выше фрагменте кода определяется функция `countClicks()`, где создается переменная `numClicks` и регистрируется обработчик событий от щелчка кнопкой мыши на всем документе. Всякий раз, когда на нем выполняется щелчок кнопкой мыши, инкрементируется значение переменной `numClicks`, а результат отображается для пользователя в окне оповещения. В данном коде обращает на себя внимание следующее.

- Внутренняя переменная `numClicks` функции `countClicks()` поддерживается активной через замыкание функции-обработчика событий от щелчка кнопкой мыши. На эту переменную можно ссылаться только в самом обработчике событий и *нигде больше*! Переменная `numClicks` изолирована от кода за пределами функции `countClicks()`. В то же время глобальное пространство имен прикладной программы не засорено переменной, которая вряд ли представляет интерес для остального кода.
- Функция `countClicks()` вызывается только в одном конкретном месте, и поэтому она определена как немедленно вызываемая (подробнее об

этом см. в главе 3), а не как функция, определяемая и далее вызываемая с помощью отдельного оператора.

Текущее состояние прикладного кода можно проанализировать в отношении того, каким образом внутренняя переменная функции (или модуля) поддерживается активной через замыкания (рис. 11.1).

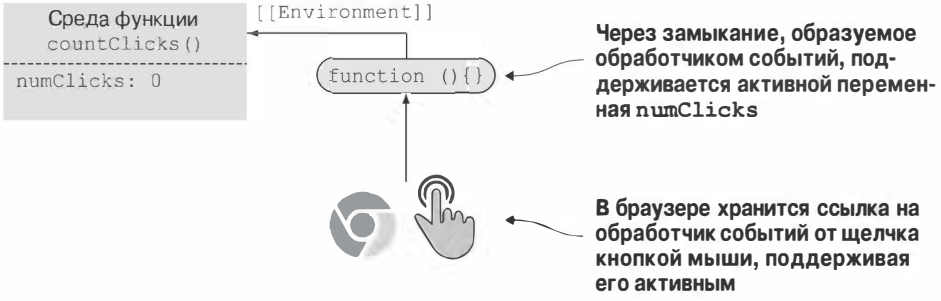


Рис. 11.1. Локальная переменная **numClicks** поддерживается активной через замыкания в обработчике событий от щелчка кнопкой мыши

А теперь, когда стало понятно, как скрывать подробности реализации модуля и поддерживать их активными столько, сколько потребуется, перейдем к рассмотрению второго минимального требования к модулям: определения их интерфейсов.

Проектный шаблон Модуль: расширение функций как модулей объектами как интерфейсами

Как правило, интерфейс модуля состоит из ряда переменных и функций, предоставляемых модулем внешнему миру. Чтобы создать такой интерфейс, проще всего воспользоваться обычным объектом в JavaScript.

В качестве примера рассмотрим создание интерфейса для упомянутого выше модуля, в котором подсчитывается количество щелчков кнопкой мыши на веб-странице, как демонстрируется в коде из листинга 11.1.

Листинг 11.1. Проектный шаблон Модуль

```
const MouseCounterModule = function() {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
```

Создать глобальную переменную модуля и присвоить ей результат выполнения немедленно вызываемой функции

Создать "закрытую" переменную модуля

Создать "закрытую" функцию модуля

Возвратить объект, представляющий интерфейс модуля. "Закрытые" переменные и функции могут быть доступны через замыкания

```
}();
```

К свойствам можно обратиться из вне через их интерфейс доступа

```
assert(typeof MouseCounterModule.countClicks === "function",
       "We can access module functionality");
assert(typeof MouseCounterModule.numClicks === "undefined"
       && typeof MouseCounterModule.handleClick === "undefined" ,
       "We cannot access internal module details");
```

При этом внутренние элементы модуля недоступны

Сначала в данном примере кода модуль реализуется с помощью немедленно вызываемой функции. В теле этой функции определяются подробности реализации модуля: одна локальная переменная `numClicks` и одна локальная функция `handleClick()`, причем они доступны только в самом модуле. Затем в данном коде создается и немедленно возвращается объект, который послужит в качестве "открытого интерфейса" модуля. Этот интерфейс содержит метод `countClicks()`, который может быть вызван за пределами модуля для доступа к его функциональным возможностям.

В то же время внутренние элементы модуля поддерживаются активными через замыкания, образуемые интерфейсом модуля, поскольку этот интерфейс становится доступным извне. Так, в методе `countClicks()` интерфейса модуля поддерживаются активными внутренние переменные модуля `numClicks` и `handleClick`, как показано на рис. 11.2.

```
const MouseCounterModule = function(){
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();
```

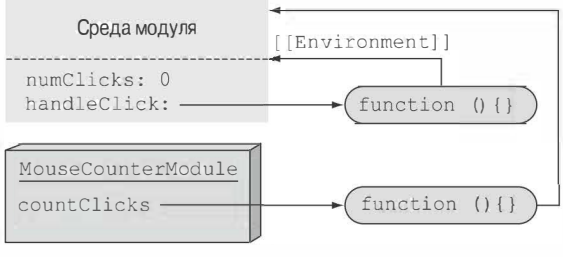


Рис. 11.2. Доступ к интерфейсу модуля через возвращаемый объект. Внутренняя реализация модуля ("закрытые" переменные и функции) поддерживается активной через замыкания, образуемые открытыми методами интерфейса

И, наконец, объект, представляющий интерфейс модуля, возвращаемого немедленно вызываемой функцией, сохраняется в переменной `MouseCounterModule`. И через нее можно легко обратиться к функциональным возможностям модуля, написав следующую строку кода:

```
MouseCounterModule.countClicks()
```

Вот, собственно, и все, что требуется знать о требованиях к реализации модуля. Воспользовавшись преимуществами немедленно вызываемых функций, можно скрыть некоторые подробности реализации модуля. А вводя объекты и замыкания в обычный объект, можно указать интерфейс модуля, раскрывающий функциональные возможности, предоставляемые модулем внешнему миру.

Такой порядок применения немедленно вызываемых функций, объектов и замыканий для создания модулей в JavaScript и составляет то, что называется проектным шаблоном `Модуль`. Этот шаблон был популяризирован Дугласом Крокфордом (Douglas Crockford) и стал одним из первых массово распространенных способов модуляризации кода JavaScript.

Таким образом, получив возможность определять модули, было бы неплохо распределить их код по нескольким файлам, чтобы можно было легче управлять ими, или же определять дополнительные функциональные возможности в существующих модулях, не видоизменяя их исходный код. Выясним далее, как это реализовать непосредственно в коде.

Расширение модулей

Попробуем теперь расширить модуль `MouseCounterModule` из предыдущего примера так, чтобы он мог дополнительно подсчитывать количество прокруток мышью. При этом первоначальный код данного модуля не должен видоизменяться, как показано в коде из листинга 11.2.

Листинг 11.2. Расширение модулей

```
const MouseCounterModule = function(){ <-- Первоначальный модуль MouseCounterModule
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();

(function(module) { <-- Немедленный вызов функции, которой в качестве аргумента передается модуль, требующий расширения
  let numScrolls = 0;
  const handleScroll = () => {
    alert(++numScrolls);
  }
```

Определить новые закрытые переменные и функции

```
  module.countScrolls = () => {
    document.addEventListener("wheel", handleScroll);
  };
})(MouseCounterModule);

assert(typeof MouseCounterModule.countClicks === "function",
       "We can access initial module functionality");
assert(typeof MouseCounterModule.countScrolls === "function",
       "We can access augmented module functionality");
```

Расширить интерфейс модуля

Передать модуль в качестве аргумента

При расширении модуля обычно используется процедура, подобная созданию нового модуля. С этой целью организуется немедленный вызов функции, но на сей раз в качестве аргумента ей передается расширяемый модуль:

```
(function(module){
  ...
  return module;
})(MouseCounterModule);
```

В теле немедленно вызываемой функции создаются все закрытые переменные и функции, требующиеся для ее выполнения. В данном случае определяются закрытая переменная и локальная функция для подсчета и сообщения о количестве прокруток:

```
let numScrolls = 0;
const handleScroll = () => {
  alert(++numScrolls);
}
```

И, наконец, модуль, доступный через параметр `module` немедленно вызываемой функции, расширяется подобно любому другому объекту следующим образом:

```
module.countScrolls = ()=> {
  document.addEventListener("wheel", handleScroll);
};
```

После выполнения этой простой операции модуль `MouseCounterModule` позволяет также подсчитывать количество прокруток документа мышью. В открытом интерфейсе этого модуля теперь имеются два метода, а самим модулем можно воспользоваться следующими способами:

```
MouseCounterModule.countClicks();
MouseCounterModule.countScrolls();
```

Метод, первоначально ставший частью интерфейса модуля

Новый метод, введенный путем расширения модуля

Как упоминалось ранее, расширение модуля происходит через немедленно вызываемую функцию аналогично созданию нового модуля. Такое расширение имеет ряд интересных побочных эффектов, связанных с замыканиями, поэтому проанализируем подробнее состояние прикладного кода после расширения модуля, как показано на рис. 11.3.

```
const MouseCounterModule = function() {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };
  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();

(function (module){
  let numScrolls = 0;
  const handleScroll = () => {
    alert(++numScrolls);
  }

  module.countScrolls = () => {
    document.addEventListener("wheel", handleClick);
  };
})(MouseCounterModule);
```

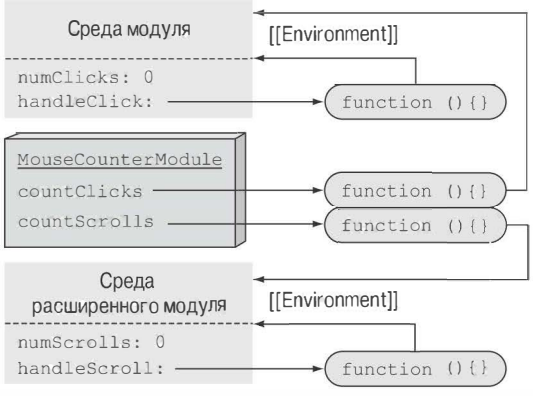


Рис. 11.3. При расширении модуля его внешний интерфейс дополняется новыми функциональными возможностями, как правило, посредством передачи модуля другой немедленно вызываемой функции. В данном примере модуль **`MouseCounterModule`** дополняется функцией **`countScrolls()`**. Обратите внимание на то, что две отдельные функции определяются в разных средах и не могут иметь доступ к внутренним переменным друг друга

Если внимательно проанализировать рис. 11.3, то можно также обнаружить одно из ограничений проектного шаблона `Модуль`, которое заключается в невозможности обращаться к закрытым переменным модуля из его расширений. Например, функция `countClicks()` поддерживает замыкание вокруг переменных `numClicks` и `handleClick`, и эти закрытые внутренние элементы модуля могут быть доступны только внутри данной функции.

К сожалению, расширение модуля функцией `countScrolls()` произведено в отдельной области видимости с совершенно новым набором закрытых переменных `numScrolls` и `handleScroll`. Функция `countScrolls()` образует замыкание только вокруг переменных `numScrolls` и `handleScroll`, поэтому ей недоступны переменные `numClicks` и `handleClick`.

На заметку

Если расширения модуля производятся через отдельные немедленно вызываемые функции, закрытые внутренние элементы модуля нельзя сделать общедоступными, поскольку каждый вызов функции приводит к созданию новой области видимости. Несмотря на это ограничение, проектный шаблон `Модуль` все же позволяет поддерживать модульность приложений на JavaScript.

Следует заметить, что в проектном шаблоне `Модуль` сами модули являются объектами как и любые другие элементы, а следовательно, их можно расширять как угодно. Например, функциональные возможности могут быть введены путем расширения объекта модуля новыми свойствами:

```
MouseCounterModule.newMethod = ()=> {...}
```

По тому же самому принципу можно без особого труда создать и подмодули:

```
MouseCounterModule.newSubmodule = () => {
  return {...};
}();
```

Однако следует иметь в виду, что все эти способы страдают одним и тем же главным недостатком, присущим проектному шаблону `Модуль`. В частности, всем последующим расширениям модуля будут недоступны определенные ранее внутренние элементы модуля.

К сожалению, у проектного шаблона `Модуль` имеются и другие недостатки. Приступая к написанию модульных приложений, следует иметь в виду, что функциональные возможности одних модулей зачастую будут зависеть от других модулей. К сожалению, проектный шаблон `Модуль` не позволяет управлять этими зависимостями. Поэтому разработчикам приходится самим упорядочивать зависимости, чтобы предоставить все необходимое для выполнения модульного кода. И если в небольших и средних приложениях это не вызывает особых трудностей, то в крупных приложениях со многими взаимозависимыми модулями в связи с этим могут возникнуть серьезные затруднения. Для разрешения подобных затруднений были созданы два конкурирующих стандарта: AMD (Asynchronous Module Definition) и CommonJS.

11.1.2. Модуляризация приложений на JavaScript по стандартам AMD и CommonJS

AMD и CommonJS — два конкурирующих стандарта определения модулей в JavaScript. Помимо некоторых синтаксических и принципиальных отличий, главное их отличие заключается в том, что стандарт AMD разработан явно с

учетом браузеров, тогда как стандарт CommonJS предназначен для универсальной среды JavaScript (например, серверов на основе Node.js) и не привязан к ограничениям, накладываемым браузерами. В этом разделе дается относительно краткий обзор обеих стандартов определения модулей, а описание их установки и включения в проекты выходит за рамки данной книги. За более подробными сведениями по данному вопросу рекомендуется обратиться к книге *JavaScript Application Design* Николя Г. Бевакуа (Nicolas G. Bevacqua; издательство Manning, 2015 г.).

Стандарт AMD

Стандарт AMD возник на основе Dojo (`https://dojotoolkit.org/`) — одного из самых распространенных наборов инструментальных средств для написания клиентских веб-приложений на JavaScript. Стандарт AMD позволяет без особого труда определять модули и их зависимости. В то же время он был изначально создан специально для браузеров. В настоящее время наиболее распространенной реализацией стандарта AMD является загрузчик модулей RequireJS (`http://requirejs.org/`).

В качестве примера рассмотрим определение небольшого модуля, у которого имеется зависимость от библиотечного модуля jQuery, как демонстрируется в коде из листинга 11.3.

Листинг 11.3. Определение модуля с зависимостью от библиотечного модуля jQuery по стандарту AMD

```
define('MouseCounterModule',['jQuery'], $ => {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      $(document).on("click", handleClick);
    }
  };
});
```

С помощью функции `define()` определяется модуль, его зависимости и фабричная функция, создающая модуль

Открытый интерфейс модуля

В стандарте AMD определена функция `define()`, которой передаются следующие аргументы.

- Идентификатор вновь созданного модуля. Этим идентификатором можно будет воспользоваться в дальнейшем, чтобы затребовать модуль из других частей системы.
- Список идентификаторов модулей, от которых зависит текущий модуль, т.е. обязательных модулей.
- Фабричная функция, инициализирующая модуль, которой передаются в качестве аргументов используемые модули.

В данном примере кода используется функция `define()` по стандарту AMD, чтобы создать модуль с идентификатором `MouseCounterModule` и зависимостью от библиотечного модуля jQuery. В силу этой зависимости по стандарту AMD функция сначала запрашивает библиотечный модуль jQuery, на что может уйти время, если файл должен быть загружен из удаленного сервера. Такое действие выполняется асинхронно во избежание блокировки. Как только все зависимости будут загружены и проинтерпретированы, вызывается фабричная функция модуля с одним аргументом для каждого запрашиваемого модуля. В данном случае указан один аргумент, поскольку новому модулю требуется только библиотечный модуль jQuery. Новый модуль создается в теле фабричной функции таким же образом, как и при использовании проектного шаблона `Модуль`, т.е. путем возврата объекта, делающего видимым открытый интерфейс модуля.

Как видите, стандарт AMD предоставляет ряд интересных преимуществ, в том числе следующие.

- Автоматическое разрешение зависимостей, избавляющее от необходимости думать о порядке, в котором следует включать модули.
- Модули могут загружаться асинхронно, благодаря чему исключаются блокировки.
- В одном файле может быть определено несколько модулей.

Итак, пояснив основной принцип стандарта AMD, перейдем к рассмотрению CommonJS – другого, не менее распространенного стандарта определения модулей.

Стандарт CommonJS

Если стандарт AMD был создан специально для браузеров, то стандарт CommonJS предназначен для определения модулей в универсальной среде JavaScript. В настоящее время он получил наибольшее распространение в сообществе пользователей платформы Node.js.

В стандарте CommonJS применяются файловая структура модулей, что позволяет сохранять модули в виде отдельных файлов. Для каждого модуля согласно стандарту CommonJS доступна переменная `module` со свойством `exports`, которое нетрудно расширить дополнительными свойствами. В конечном счете содержимое свойства `module.exports` используется в качестве открытого интерфейса модуля.

Если требуется воспользоваться модулем в других частях приложения, его сначала нужно запросить. Файл модуля будет загружен синхронно, после чего станет доступен его открытый интерфейс. Именно поэтому стандарт CommonJS чаще всего используется на стороне сервера, где модули загружаются относительно быстро, поскольку для этого достаточно лишь выполнить операцию чтения в файловой системе, а не на стороне клиента, где модули при-

ходится загружать из удаленного сервера и их синхронная загрузка зачастую приводит к блокировке.

В качестве примера рассмотрим снова определение модуля `MouseCounterModule`, но на этот раз по стандарту CommonJS, как демонстрируется в коде из листинга 11.4.

Листинг 11.4. Определение модуля по стандарту CommonJS

```
// MouseCounterModule.js
const $ = require("jQuery");
let numClicks = 0;
const handleClick = () => {
  alert(++numClicks);
};

module.exports = {
  countClicks: () => {
    $(document).on("click", handleClick);
  }
};
```

Запрос на синхронную загрузку библиотечного модуля jQuery

Изменить свойство `module.exports`, чтобы указать открытый интерфейс модуля

Чтобы включить модуль в другой файл, достаточно написать следующий фрагмент кода:

```
const MouseCounterModule = require("MouseCounterModule.js");
MouseCounterModule.countClicks();
```

Как видите, все очень просто. Принцип действия стандарта CommonJS предполагает распределение модулей по отдельным файлам, и поэтому любой код, размещаемый в файловом модуле, становится частью этого модуля. Следовательно, отпадает необходимость заключать переменные в оболочку немедленно вызываемых функций. Все переменные, определяемые в модуле, безопасно содержатся в области видимости текущего модуля, и никак не влияют на глобальную область видимости. Например, переменные (`$`, `numClicks` и `handleClick`) оказываются в области видимости рассматриваемого здесь модуля, несмотря на то, что они определены в коде верхнего уровня (т.е. за пределами всех функций и блоков), что, по существу, делает их глобальными переменными в стандартных файлах JavaScript.

И в этом случае очень важно заметить, что из внешнего модуля доступны только переменные и функции, видимые через свойство объекта `module.exports`. Сама процедура — такая же, как и при использовании проектного шаблона `Модуль`, только вместо возврата совершенно нового объекта в среде заранее создается объект, который может быть расширен новыми методами и свойствами интерфейса модуля.

Определение модулей по стандарту CommonJS имеет следующие преимущества.

- Простой синтаксис. Достаточно указать только свойства `module.exports`, оставив остальную часть модуля практически такой же, как

и при написании стандартного кода JavaScript. Запрос модулей также осуществляется очень просто, для чего достаточно вызвать функцию `require()`.

- Формат CommonJS является стандартным для платформы Node.js. Благодаря этому тысячи пакетов становятся доступными через утилиту `npm` — диспетчер пакетов среды Node.js.

Самый крупный недостаток стандарта CommonJS заключается в том, что он разработан без учета среды браузеров. В интерпретаторе JavaScript браузера отсутствует поддержка переменной `module` и свойства `export`, и поэтому модули, определяемые по стандарту CommonJS, приходится упаковывать в формате, удобочитаемом для браузера. И этого можно добиться с помощью инструментального средства Browserify (`http://browserify.org/`) или RequireJS (`http://requirejs.org/docs/commonjs.html`).

Наличие стандартов AMD и CommonJS для определения модулей привело к разделению разработчиков на два (иногда противоположных) лагеря. Если речь идет о работе над относительно закрытыми проектами, то достаточно выбрать наиболее подходящий стандарт. Но трудности могут появиться, когда потребуется повторное использование кода из противоположного лагеря, и тогда придется преодолеть немало препятствий. В качестве выхода из столь затруднительного положения можно, например, воспользоваться проектным шаблоном `UMD` (Universal Module Definition — универсальное определение модулей; `https://github.com/umdjs/umd`) с несколько замысловатым синтаксисом, позволяющим применять один и тот же модульный файл согласно обоим стандартам, AMD и CommonJS. Рассмотрение этого вопроса выходит за рамки данной книги, но если он заинтересует вас, то в Интернете вы сможете найти немало полезных ресурсов.

Правда, комитет ECMAScript, отвечающий за стандартизацию JavaScript, признал необходимость поддержки единообразного синтаксиса модулей во всех средах JavaScript. И с этой целью в ES6 был определен новый стандарт поддержки модулей, который призван окончательно разрешить все упомянутые выше недоразумения.

11.2. Модули по стандарту ES6

В ES6 модули определяется таким образом, чтобы можно было использовать преимущества обоих стандартов, AMD и CommonJS. В частности,

- подобно стандарту CommonJS, модули обладают относительно простым синтаксисом и распределяются по отдельным файлам;
- подобно стандарту AMD, модули поддерживают асинхронную загрузку.

На заметку

Встроенные модули являются частью стандарта ES6. Как станет ясно в дальнейшем, в синтаксис определения модулей, принятый в стандарте ES6, включена дополнительная семантика и ключевые слова (например, **export** и **import**), которые пока еще не поддерживаются в современных браузерах. Чтобы пользоваться модулями в настоящее время, придется выполнить транспиляцию модульного кода средствами Traceur (**https://github.com/google/traceur-compiler**), Babel (**http://babeljs.io/**) или TypeScript (**www.typescriptlang.org/**). Можно также воспользоваться библиотекой SystemJS (**https://github.com/systemjs/systemjs**), обеспечивающей поддержку загрузки модулей по всем ныне действующим стандартам: AMD, CommonJS и даже ES6. Подробные инструкции по применению библиотеки SystemJS можно найти в хранилище данного проекта (**https://github.com/systemjs/systemjs**).

Принцип определения модулей по стандарту ES6 основывается на том, что из модуля явным образом экспортируются только идентификаторы, доступные за его пределами. А все остальные идентификаторы, даже те, что определены в области видимости верхнего уровня (т.е. глобальной области видимости в стандартном коде JavaScript), доступны только в самом модуле. И такое решение было навеяно стандартом CommonJS.

Чтобы обеспечить подобные функциональные возможности, в стандарт ES6 были введены следующие ключевые слова:

- **export** – для доступа к определенным идентификаторам за пределами модуля;
- **import** – для импорта экспортированных идентификаторов модулей.

Синтаксис экспорта и импорта функциональных возможностей модулей довольно прост. Но у него имеется немало едва заметных особенностей, которые мы постепенно рассмотрим далее.

11.2.1. Функциональные возможности экспорта и импорта

Начнем с простого примера кода из листинга 11.5, где демонстрируется, каким образом функциональные возможности экспортируются из одного модуля и импортируются в другой.

Листинг 11.5. Экспорт из библиотечного модуля Ninja.js

```
const ninja = "Yoshi";
export const message = "Hello";

export function sayHiToNinja() {
  return message + " " + ninja;
}
```

Определить в модуле переменную верхнего уровня

Определить переменную и функцию и экспортировать их из модуля с помощью ключевого слова export

Получить доступ к внутренней переменной модуля через открытый интерфейс API

Сначала в данном примере кода определяется переменная `ninja`, доступная только в самом модуле, хотя это и сделано в коде верхнего уровня (в коде до стандарта ES6 такая переменная стала бы глобальной).

Затем в данном примере кода определяется еще одна переменная `message` верхнего уровня, доступная за пределами модуля благодаря ключевому слову `export`. И, наконец, создается и экспортируется функция `sayHiToNinja()`.

Вот и все! Это минимальный синтаксис, который следует знать для определения модулей. Чтобы экспортировать функциональные возможности из модуля, не нужно вызывать промежуточные функции или запоминать какой-нибудь замысловатый синтаксис. Достаточно написать код в привычном для JavaScript стиле, указав лишь перед некоторыми идентификаторами (например, переменных, функций или классов) ключевое слово `export`.

Прежде чем выяснять, каким образом импортируются функциональные возможности модулей, рассмотрим другой способ экспорта идентификаторов. Все, что требуется экспортировать из модуля, перечисляется в его конце, как выделено полужирным шрифтом в примере кода из листинга 11.6.

Листинг 11.6. Экспорт в конце модуля

```
const ninja = "Yoshi";
const message = "Hello";

function sayHiToNinja() {
  return message + " " + ninja;
}

export { message, sayHiToNinja };
```

Определить все идентификаторы модуля

Экспортировать некоторые идентификаторы модуля

Такой способ экспорта идентификаторов модуля чем-то напоминает проектный шаблон `Модуль`, где немедленно вызываемая функция возвращает объект, представляющий открытый интерфейс данного модуля. И особенно он напоминает стандарт CommonJS, где открытый интерфейс модуля расширяет свойство объекта `module.exports`.

Но независимо от способа экспорта идентификаторов определенного модуля, для их импорта придется воспользоваться ключевым словом `import`, как выделено полужирным шрифтом в примере кода из листинга 11.7.

Листинг 11.7. Импорт из модуля Ninja.js

```
import { message, sayHiToNinja} from "Ninja.js";

assert(message === "Hello",
       "We can access the imported variable");
assert(sayHiToNinja() === "Hello Yoshi",
       "We can say hi to Yoshi from outside the module");

assert(typeof ninja === "undefined",
       "But we cannot access Yoshi directly");
```

Импортировать привязку идентификатора из модуля с помощью ключевого слова `import`

Теперь можно обратиться к импортированной переменной и вызвать импортированную функцию

Неэкспортированные переменные недоступны напрямую

В данном примере кода ключевое слово `import` применяется для импорта переменной `message` и функции `sayHiToNinja()` из модуля `ninja` следующим образом:

```
import { message, sayHiToNinja} from "Ninja.js";
```

Таким образом, мы получаем доступ к этим двум идентификаторам, определенным в модуле `ninja`. И, наконец, возможность доступа к переменной `message` и функции `sayHiToNinja()` проверяется в следующем фрагменте кода:

```
assert(message === "Hello",
       "We can access the imported variable");
assert(sayHiToNinja() === "Hello Yoshi",
       "We can say hi to Yoshi from outside the module");
```

Но недоступными оказываются переменные, которые не были экспортированы и импортированы. Например, переменная `ninja` недоступна потому, что ее определение не помечено ключевым словом `export`, что и подтверждается в приведенном ниже фрагменте кода.

```
assert(typeof ninja === "undefined",
       "But we cannot access Yoshi directly");
```

Благодаря поддержке модулей в стандарте ES6 наконец-то появилась возможность меньше злоупотреблять глобальными переменными. Все, что не помечено явно ключевым словом `export`, остается надежно изолированным в модуле.

В данном примере использован *именованный экспорт*, позволяющий экспортировать несколько идентификаторов из модуля, как это было сделано с идентификаторами `message` и `sayHiToNinja`. Но в связи с возможностью экспортировать большое количество идентификаторов перечислять их полностью в операторе импорта не очень удобно. Поэтому для этой цели употребляется сокращенное обозначение, как демонстрируется в примере кода из листинга 11.8.

Листинг 11.8. Импорт всего именованного экспорта из модуля Ninja.js

```
import * as ninjaModule from "Ninja.js";
```

Импортировать все экспортированные идентификаторы, используя сокращенное обозначение *

```
assert(ninjaModule.message === "Hello",
       "We can access the imported variable");
assert(ninjaModule.sayHiToNinja() === "Hello Yoshi",
       "We can say hi to Yoshi from outside the module");
```

Обратиться к именованному экспорту, используя синтаксис свойства

```
assert(typeof ninjaModule.ninja === "undefined",
       "But we cannot access Yoshi directly");
```

Неэкспортированные идентификаторы по-прежнему недоступны

Как показано в коде из листинга 11.8, для импорта всех экспортированных идентификаторов из модуля употребляется сокращенная запись `import *` в сочетании с идентификатором, который затем будет использоваться для ссылки

на весь модуль (в данном случае это идентификатор `ninjaModule`). После этого к экспортированным идентификаторам можно обращаться, используя синтаксис свойства, например: `ninjaModule.message`, `ninjaModule.sayHiToNinja`. Следует, однако, иметь в виду, что те переменные верхнего уровня модуля, которые не были экспортированы, по-прежнему остаются недоступными, (например, переменная `ninja`).

Экспорт по умолчанию

Нередко вместо экспорта из модуля взаимосвязанных идентификаторов, нужно представить весь модуль в виде единственной операции экспорта. Нечто подобное нередко происходит в том случае, если модули содержат единственный класс, как показано в примере кода из листинга 11.9.

Листинг 11.9. Экспорт по умолчанию из модуля Ninja.js

```
export default class Ninja {
  constructor(name){
    this.name = name;
  }
}

export function compareNinjas(ninja1, ninja2){
  return ninja1.name === ninja2.name;
}
```

Указать привязку к модулю по умолчанию, используя ключевые слова `export` и `default`

Наряду с экспортом по умолчанию можно по-прежнему пользоваться именованным экспортом

В данном примере кода после ключевого слова `export` указывается ключевое слово `default`, обозначающее привязку к модулю по умолчанию. Такой привязкой в данном случае служит класс `Ninja`. Но несмотря на указание привязки по умолчанию, для экспорта дополнительных идентификаторов можно по-прежнему пользоваться именованным экспортом, как это было показано на примере функции `compareNinjas()`.

И теперь для импорта функциональных возможностей из модуля Ninja.js можно воспользоваться упрощенным синтаксисом (выделено полужирным в примере кода из листинга 11.10).

Листинг 11.10. Импорт экспорта по умолчанию

Чтобы импортировать экспорт по умолчанию, не нужно указывать фигурные скобки; при этом можно использовать любое удобное имя

Именованный экспорт можно по-прежнему импортировать

```
import ImportedNinja from "Ninja.js";
import {compareNinjas} from "Ninja.js";

const ninja1 = new ImportedNinja("Yoshi");
const ninja2 = new ImportedNinja("Hattori");

assert(ninja1 !== undefined
       && ninja2 !== undefined, "We can create a couple of Ninjas");
```

Создать пару объектов ниндзя и проверить их существование

```
assert(!compareNinjas(ninja1, ninja2),
       "We can compare ninjas");
```

Именованный экспорт по-прежнему доступен

Сначала в данном примере кода импортируется экспорт по умолчанию. С этой целью используется более простой синтаксис без фигурных скобок, которые обязательны для импорта именованного экспорта. Следует также заметить, что для обозначения экспорта по умолчанию можно выбрать произвольное имя, и это совсем не обязательно должно быть имя, использованное при экспорте. В данном примере именем `ImportedNinja` обозначается класс `Ninja`, определенный в модуле Ninja.js.

Далее в рассматриваемом здесь примере кода импортируется именованный элемент, чтобы продемонстрировать, как и в предыдущих примерах, возможность как экспорта по умолчанию, так и именованного экспорта элементов из одного и того же модуля. Наконец, в данном примере кода получаются два экземпляра объекта `ninja` и вызывается функция `compareNinjas()`, чтобы убедиться, что весь импорт действует должным образом.

Поскольку в данном случае оба элемента импортированы из одного и того же файла, в стандарте ES6 реализован специальный сокращенный синтаксис, как показано ниже.

```
import ImportedNinja, {compareNinjas} from "Ninja.js";
```

Здесь в одном операторе импорта из модуля Ninja.js через запятую (`,`) указывается как экспорт по умолчанию, так и именованный экспорт.

Переименование экспорта и импорта

При необходимости экспорт и импорт можно переименовать. Начнем с переименования экспорта, как в приведенном ниже примере кода, где в комментариях указана принадлежность кода к определенному файлу.

```
//************* Greetings.js ************/
function sayHi(){
  return "Hello";
}
```

Определить функцию `sayHi()`

```
assert(typeof sayHi === "function"
       && typeof sayHello === "undefined",
       "Within the module we can access only sayHi");
```

Проверить возможность доступа к функции `sayHi()` только по идентификатору, но не по псевдониму!

```
export { sayHi as sayHello }
```

Указать псевдоним идентификатора с помощью ключевого слова `as`

```
//************* main.js *****************/
import {sayHello } from "Greetings.js";

assert(typeof sayHi === "undefined"
       && typeof sayHello === "function",
       "When importing, we can only access the alias");
```

При импорте доступен только псевдоним `sayHello`

В приведенном выше примере кода определяется функция `sayHi()` и проверяется возможность доступа к ней только по идентификатору `sayHi`, но не по псевдониму `sayHello`, который определяется в конце модуля с помощью ключевого слова `as`:

```
export { sayHi as sayHello }
```

Переименовать экспорт можно только в такой форме, а не предваряя объявление функции или переменной ключевым словом `export`. А когда выполняется импорт переименованного экспорта, то обращение к импорту осуществляется по указанному псевдониму:

```
import { sayHello } from "Greetings.js";
```

И, наконец, возможность доступа к импортированной функции по псевдониму, но не по первоначальному ее идентификатору проверяется следующим образом:

```
assert(typeof sayHi === "undefined"
       && typeof sayHello === "function",
       "When importing, we can only access the alias");
```

Аналогичная ситуация возникает при переименовании импорта, как в следующем примере кода:

```
/************* Hello.js *************/
export function greet(){
  return "Hello";
}

/************* Salute.js *************/
export function greet(){
  return "Salute";
}

/************* main.js *************/
import { greet as sayHello } from "Hello.js";
import { greet as salute } from "Salute.js";

assert(typeof greet === "undefined",
       "We cannot access greet");

assert(sayHello() === "Hello" && salute() === "Salute",
       "We can access aliased identifiers!");
```

Экспортировать функцию под именем `greet()` из модуля `Hello.js`

Экспортировать функцию под тем же самым именем `greet()` из модуля `Salute.js`

Указать псевдонимы импорта с помощью ключевого слова `as`, исключив конфликт имен

Функция недоступна по первоначальному имени

Но она доступна через псевдонимы

Как и при экспорте идентификаторов, для указания псевдонимов при импорте идентификаторов из других модулей можно воспользоваться ключевым словом `as`. Это удобно в том случае, если требуется предоставить имя, более подходящее для текущего контекста, или избежать конфликта имен, как и в приведенном выше небольшом примере кода.

На этом исследование синтаксиса определения модулей по стандарту ES6 завершается. А краткие его итоги подведены в табл. 11.1.

Таблица 11.1. Краткий обзор синтаксиса определения модулей по стандарту ES6

Код	Назначение
`export const ninja = "Yoshi";`	Экспорт именованной переменной
`export function compare(){}`	Экспорт именованной функции
`export class Ninja{}`	Экспорт именованного класса
`export default class Ninja{}`	Экспорт класса по умолчанию
`export default function Ninja(){}`	Экспорт функции по умолчанию

`const ninja = "Yoshi";`
`function compare(){};`
`export {ninja, compare};`
	Экспорт существующих переменных

`export {ninja as samurai, compare};`	Экспорт переменной по новому имени
`import Ninja from "Ninja.js";`	Импорт экспорта по умолчанию
`import {ninja, Ninja} from "Ninja.js";`	Импорт именованного экспорта
`import * as Ninja from "Ninja.js";`	Импорт всего именованного экспорта из модуля
`import {ninja as iNinja} from "Ninja.js";`	Импорт именованного экспорта по новому имени

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Крупные и монолитные кодовые базы намного более трудны для понимания и сопровождения, чем более мелкие и хорошо организованные кодовые базы. Чтобы усовершенствовать структуру и организацию прикладных программ, можно, в частности, разбить их на более мелкие, слабо связанные части, или модули.
- Модули являются более крупными единицами организации кода, чем объекты и функции. Они позволяют разделять прикладные программы на принадлежащие друг другу части.
- В общем, модули помогают лучше понимать, проще сопровождать исходный код и усовершенствовать его повторное использование.
- До принятия стандарта ES6 в языке JavaScript отсутствовали встроенные модули, поэтому разработчикам приходилось творчески подходить к применению имеющихся языковых средств JavaScript, чтобы добиться модуляризации прикладного кода. Одним из самых распространенных

способов создания модулей стало сочетание немедленно вызываемых функций с замыканиями.

  - Немедленно вызываемые функции применяются потому, что они образуют новую область видимости для определения в модуле переменных, недоступных за пределами этой области видимости.
  - Замыкания применяются потому, что они позволяют поддерживать активными переменные в модуле.
  - Наибольшее распространение при этом нашел проектный шаблон `Модуль`, где вызов немедленно вызываемой функции обычно сочетается с возвратом нового объекта, представляющего открытый интерфейс модуля.

- Помимо проектного шаблона `Модуль`, существуют два распространенных стандарта: AMD (Asynchronous Module Definition) и CommonJS. Первый из них предназначен для поддержки модулей в браузерах, а второй чаще всего применяется в серверных приложениях JavaScript.
  - Стандарт AMD позволяет автоматически разрешать зависимости, асинхронно загружать модули, исключая тем самым блокировку.
  - Стандарт CommonJS обладает простым синтаксисом, позволяет синхронно загружать модули и поэтому в большей степени пригоден для применения на стороне сервера. А с помощью диспетчера пакетов Node.js `npm` можно сделать доступными многие модули.
- Модули, введенные в стандарт ES6, призваны вобрать в себя все преимущества стандартов AMD и CommonJS. Такие модули обладают простым синтаксисом, навеянным стандартом CommonJS, а также допускают асинхронную загрузку, как предусмотрено в стандарте AMD.
  - Модули в стандарте ES6 распределяются по отдельным файлам.
  - Идентификаторы можно экспортировать, используя ключевое слово `export`, чтобы обращаться к другим модулям.
  - Идентификаторы можно импортировать из других модулей, используя ключевое слово `import`.
  - У модуля может быть единственный экспорт по умолчанию (`default`), с помощью которого можно представить модуль в целом.
  - Экспорт и импорт модулей можно переименовывать с помощью ключевого слова `as`.

Упражнения

1. Какой из перечисленных ниже механизмов допускает наличие закрытых переменных модуля в проектном шаблоне `Модуль`?

   а) Прототипы

б) Замыкания

в) Обещания

2. В приведенном ниже фрагменте кода применяются модули, внедренные в стандарт ES6. Какие из перечисленных ниже идентификаторов могут быть доступны, если импортировать модуль?

```
const spy = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
export const general = "Minamoto";
```

а) `spy`

б) `command`

в) `general`

3. В приведенном ниже фрагменте кода применяются модули, внедренные в стандарт ES6. Какие из перечисленных ниже идентификаторов могут быть доступны, если импортировать модуль?

```
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general = "Minamoto";

export {ninja as spy};
```

а) `spy`

б) `command`

в) `general`

г) `ninja`

4. Какой из перечисленных ниже видов импорта модулей допустим?

```
// Файл модуля: personnel.js
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general = "Minamoto";

export {ninja as spy};
```

а) `import {ninja, spy, general} from "personnel.js"`

б) `import * as Personnel from "personnel.js"`

в) `import {spy} from "personnel.js"`

5. Какой из перечисленных ниже операторов импортирует класс `Ninja` в приведенном ниже фрагменте кода?

```
// Ninja.js
export default class Ninja {
  skulk(){ return "skulking"; }
}
```

а) `import Ninja from "Ninja.js"`

б) `import * as Ninja from "Ninja.js"`

в) `import * from "Ninja.js"`

Часть IV

Исследование браузеров

Изучив основные языковые средства JavaScript, перейдем к браузерам — среде, в которой выполняется большинство приложений на JavaScript.

В главе 12 мы подробно рассмотрим модель DOM. В ней будут представлены эффективные методики видоизменения этой модели и достижения высокого быстродействия и динамичности веб-приложений.

В главе 13 речь пойдет о событиях. Особое внимание в ней будет уделено циклу ожидания событий и его влиянию на воспринимаемую производительность веб-приложений.

И в последней главе данной книги обсуждаются не очень приятные, но важные вопросы кросс-браузерной разработки. И хотя за последние годы положение дел в этой области заметно улучшилось, тем не менее, до сих пор нет никаких гарантий, что прикладной код будет одинаково работать в каждом браузере. Поэтому в главе 14 представлены стратегии кросс-браузерной разработки веб-приложений.

12 Работа с моделью DOM

В этой главе...

- Вставка HTML-кода в объектную модель документа (DOM)
- Представление об атрибутах и свойствах модели DOM
- Исследование вычисляемых стилей
- Преодоление перегрузки верстки веб-страниц

До сих пор речь шла в основном о языке JavaScript. И хотя у самого языка JavaScript имеется немало особенностей, разработка веб-приложений определенно не облегчается за счет внедрения объектной модели документа (Document Object Model — DOM). Одним из главных средств для написания высокодинамичных веб-приложений, оперативно реагирующих на действия пользователей, служит видоизменение модели DOM. Но если открыть любую библиотеку JavaScript, то в ней, как ни странно, можно обнаружить немало фрагментов длинного и сложного кода для выполнения простых операций в модели DOM. Даже такие простые на первый взгляд операции, как клонирование или удаление узла (им соответствуют методы `cloneNode()` и `removeChild()` в модели DOM), реализуются относительно сложно. И в связи с этим возникают следующие вопросы:

- Почему этот код такой сложный?
- Зачем разбираться в нем, если библиотека возьмет на себя все хлопоты по выполнению нужных операций?

Основной побудительной причиной для оптимизации такого кода служит *производительность*. Имея ясное представление о том, как в библиотеках реализуется видоизменение, или модификация модели DOM, вы сможете написать код, который будет работать лучше и быстрее, обращаясь к библиотеке. А кроме того, у вас появится возможность применить в своем коде специальные приемы из библиотеки.

В начале этой главы будет показано, каким образом благодаря внедрению произвольного HTML-кода динамически создаются новые части веб-страниц. Затем будут исследованы все препятствия, которые браузеры ставят перед разработчиками веб-приложений на JavaScript в отношении свойств элементов и атрибутов разметки. В ходе этого исследования будут раскрыты причины, по которым результаты не всегда оказываются такими, как предполагалось.

Это же относится и к каскадным таблицам стилей (Cascading Style Sheets — CSS) и стилевому оформлению элементов разметки. Многие трудности, возникающие при разработке динамического веб-приложения, порождены сложностями установки и получения стилевого оформления элементов разметки. В одной главе невозможно охватить все, что требуется знать о том, как обращаться со стилевым оформлением элементов разметки, поскольку данной теме можно посвятить отдельную книгу, но самые основы здесь все же рассматриваются.

И в завершение главы будут рассмотрены трудные вопросы производительности, которые могут возникнуть в том случае, если не уделить должного внимания порядку видоизменения и чтения информации из модели DOM. Итак, начнем с того, как вставлять HTML-код на веб-страницы.

Знаете ли вы?

Зачем требуется предварительный синтаксический анализ самозакрывающихся элементов HTML-кода на веб-странице перед их вставкой?

Каковы преимущества работы с фрагментами модели DOM при вставке HTML-кода?

Как определить размеры скрытого элемента разметки на странице?

12.1. Вставка HTML-кода в объектную модель документа

В этом разделе рассматривается эффективный способ вставки HTML-кода, представленный символьной строкой, в произвольное место документа. Этот конкретный способ выбран потому, что он зачастую применяется для создания высокодинамичных веб-страниц, где пользовательский интерфейс видоизменяется в ответ на действия пользователя или поступление данных от сервера. Он оказывается особенно полезным в следующих случаях.

- Вставка произвольного HTML-кода на странице, ввод шаблонов и манипулирование ими на стороне клиента.
- Извлечение и вставка HTML-содержимого, получаемого от сервера.

Правильная реализация этих функциональных возможностей вызывает определенные технические трудности, особенно в сравнении с созданием интерфейса API, который предназначен для конструирования модели DOM в объектно-ориентированном стиле и реализуется проще, хотя и требует более высокого уровня абстракции, чем вставка HTML-кода. Рассмотрим следующий пример создания элементов разметки из символьной строки, заданной в формате HTML, средствами библиотеки jQuery:

```
$(document.body).append(
    "<div><h1>Greetings</h1><p>Yoshi here</p></div>")
```

А теперь сравните это со следующим способом, в котором применяется интерфейс DOM API:

```
const h1 = document.createElement("h1");
h1.textContent = "Greetings";

const p = document.createElement("p");
p.textContent = "Yoshi here";

const div = document.createElement("div");

div.appendChild(h1);
div.appendChild(p);

document.body.appendChild(div);
```

Какой же из них лучше?

Чтобы ответить на этот вопрос мы реализуем сызнова собственный метод работы с моделью DOM. Его реализация разделяется на следующие этапы.

1. Преобразование произвольной, но корректной символьной строки в HTML формате в структуру DOM.
2. Вставка этой структуры DOM в произвольном месте документа как можно более эффективным образом.

В результате выполнения этих этапов разработчики веб-страниц получат в свое распоряжение изящный способ вставки HTML-кода в документ. Приступим к его реализации.

12.1.1. Преобразование из формата HTML в структуру DOM

В преобразовании символьной строки формата HTML (в дальнейшем просто HTML-строки) в структуру DOM нет ничего таинственного. Для этой цели мы воспользуемся хорошо известным вам средством: свойством `innerHTML` элементов модели DOM. Процесс преобразования выполняется в несколько этапов.

1. Проверка корректности HTML-кода, содержащегося в HTML-строке.
2. Заключение этой строки в объемлющую разметку, как того требуют правила в браузерах.

3. Вставка HTML-строки с помощью свойства `innerHTML` в фиктивный элемент модели DOM.
4. Извлечение узлов модели DOM в обратном порядке.

Эти этапы не очень сложны, за исключением самой вставки, таящей в себе некоторые скрытые трудности. Тем не менее они вполне преодолимы. Рассмотрим все эти этапы по очереди.

Предварительная обработка исходной HTML-строки

Прежде всего необходимо привести исходный HTML-код в соответствующий порядок. Рассмотрим в качестве примера приведенный ниже эскиз HTML-кода, позволяющего выбрать ниндзя (с помощью элемента разметки `option`) и отображающий подробные сведения о выбранном ниндзя в таблице. Эти сведения предполагается выводить в дальнейшем.

```
<option>Yoshi</option>
<option>Kuma</option>
<table/>
```

Такой HTML-код вызывает две проблемы. Во-первых, элементы разметки `option` не должны стоять обособленно. Если следовать надлежащей семантике языка HTML, то они должны содержаться в контейнере `select`. И во-вторых, несмотря на то, что в языках разметки обычно допускаются самозакрывающиеся элементы вроде `<table />`, не содержащие дочерних элементов, в HTML самозакрытие распространяется лишь на небольшие подмножества элементов разметки, к числу которых элемент `table` не относится. Всякая попытка воспользоваться подобным синтаксисом в других случаях, скорее всего, вызовет осложнения в работе некоторых браузеров.

Начнем с разрешения трудностей, связанных с самозакрывающимися элементами разметки. Для преобразования таких элементов разметки, как `"<table />"`, в элементы `"<table></table>"`, единообразно обрабатываемые во всех браузерах, можно выполнить быстрый предварительный синтаксический анализ HTML-строки, как показано в примере кода из листинга 12.1.

Листинг 12.1. Проверка правильности интерпретации самозамыкающихся элементов разметки

```
const tags =
/^(area|base|br|col|embed|hr|img|input|keygen|link|menuitem
      |meta|param|source|track|wbr)$/i;
function convert(html) {
  return html.replace(
      /(<(\w+)[^>]*?)\/>/g, (all, front, tag) => {
    return tags.test(tag) ? all :
    front + "></" + tag + ">";
  });
}
```

Воспользоваться регулярным выражением для поиска тех дескрипторов, которые можно игнорировать

Функция, в которой регулярные выражения используются для преобразования самозакрывающихся дескрипторов в обычную форму

```
assert(convert("<a/>") === "<a></a>", "Check anchor conversion.");
assert(convert("<hr/>") === "<hr/>", "Check hr conversion.");
```

В результате применения функции `convert()` к HTML-строке в данном примере кода получается следующий HTML-код:

```
<option>Yoshi</option>
<option>Kuma</option>
<table/></table>
```

По завершении первого этапа у нас по-прежнему осталась одна проблема, поскольку элементы разметки `option` не содержатся в контейнере `select`. Поэтому необходимо каким-то образом выяснить, требуется ли вообще заключать элементы разметки в контейнер.

Заключение HTML-кода в контейнер

В соответствии с семантикой HTML некоторые элементы HTML-кода должны быть заключены в соответствующий контейнер перед их вставкой в DOM. Например, элемент разметки `<option>` должен быть заключен в контейнер `<select>`. Эту задачу можно решить двумя способами, и оба требуют составления определенной схемы соответствия проблематичных элементов разметки их контейнерам.

- HTML-строку можно вставить с помощью свойства `innerHTML` непосредственно в конкретный родительский элемент, созданный ранее с помощью метода `document.createElement()`. И хотя такой способ вполне может работать в некоторых браузерах, его универсальность совсем не гарантируется для всех браузеров.
- HTML-строку можно заключить в подходящий контейнер, а затем вставить непосредственно в любой родительский элемент (например, `div`). Это более надежный, но и трудоемкий способ.

Второй способ предпочтительнее первого. Он требует написания очень малого количества специфичного для браузера кода, в отличие от первого способа, который предполагает написание кода, практически полностью зависящего от браузера.

Проблематичные элементы, которые требуется заключить в соответствующие контейнеры, вполне поддаются управлению и могут быть разделены на семь групп. В табл. 12.1 многоточие (`...`) обозначает место, где должны быть вставлены элементы разметки.

Таблица 12.1. Элементы разметки, которые должны содержаться в контейнерах

Имя элемента	Родительский элемент
`<option>`, `<optgroup>`	`<select multiple>...</select>`
`<legend>`	`<fieldset>...</fieldset>`

Окончание табл. 12.1

Имя элемента	Родительский элемент
`<thead>`, `<tbody>`, `<tfoot>`, `<colgroup>`, `<caption>`	`<table>...</table>`

`<tr>`	`<table><thead>...</thead></table>`
	`<table><tbody>...</tbody></table>`
	`<table><tfoot>...</tfoot></table>`

`<td>`, `<th>`	`<table><tbody><tr>...</tr></tbody></table>`

`<col>`	`<table>`
	`  <tbody></tbody>`
	`  <colgroup>...</colgroup>`
	`</table>`

Приведенная выше таблица не требует особых пояснений, за исключением следующих моментов.

- Вместо элемента одиночного выбора используется элемент разметки `<select>` с атрибутом `multiple`, поскольку в нем не производится автоматическая проверка любых вариантов выбора, которые в нем размещены, тогда как в элементе одиночного выбора автоматически проверяется первый вариант выбора.
- В элемент разметки `<col>` входит дополнительный элемент `<tbody>`, без которого элемент разметки `<colgroup>` сформируется неправильно.

Итак, поместив элементы разметки в соответствующие контейнеры, перейдем к их формированию. Используя информацию из табл. 12.1, можно сформировать HTML-код, который требуется вставить в элемент модели DOM, как демонстрируется в примере кода из листинга 12.2.

Листинг 12.2. Формирование узлов модели DOM из исходной разметки

```
function getNodes(htmlString, doc) {
  const map = {
    "<td":[3,"<table><tbody><tr>","</tr></tbody></table>"],
    "<th":[3,"<table><tbody><tr>","</tr></tbody></table>"],
    "<tr":[2,"<table><thead>","</thead></table>"],
    "<option":[1,"<select multiple>","</select>"],
    "<optgroup":[1,"<select multiple>","</select>"],
    "<legend":[1,"<fieldset>","</fieldset>"],
    "<thead":[1,"<table>","</table>"],
    "<tbody":[1,"<table>","</table>"],
    "<tfoot":[1,"<table>","</table>"],
    "<colgroup":[1,"<table>","</table>"],
    "<caption":[1,"<table>","</table>"],
    "<col":[2,"<table><tbody></tbody><colgroup>","</colgroup></table>"],
  };
  const tagName = htmlString.match(/<\w+/);
  let mapEntry = tagName ? map[tagName[0]] : null;
```

Отображение элементов разметки и требующихся для них контейнеров. Для каждого элемента указано количество дескрипторов в новом узле, а также код открывающих и закрывающих дескрипторов

Совпадение с открывающей угловой скобкой и именем дескриптора вставляемого элемента разметки

```
  if (!mapEntry) { mapEntry = [0, " "," " ];}
  let div = (doc || document).createElement("div");
  div.innerHTML = mapEntry[1] + htmlString + mapEntry[2];
  while (mapEntry[0]--) { div = div.lastChild;}
  return div.childNodes;
}
assert(getNodes("<td>test</td><td>test2</td>").length === 2,
       "Get two nodes back from the method.");
assert(getNodes("<td>test</td>")[0].nodeName === "TD",
       "Verify that we're getting the right node.");
```

Если элемент разметки присутствует в отображении, извлечь его, а иначе построить фиктивный элемент разметки с пустой "родительской" разметкой и нулевым числом дескрипторов

Создать элемент разметки , а в нем — новые узлы. Для этого используется передаваемый документ, если он существует, а иначе — текущий документ

Возвратить вновь созданный элемент разметки

Пройти вновь созданное дерево на глубину, указанную в элементе отображения. Это должен быть родительский элемент нужного узла, созданного из разметки

Заключить входящую разметку в родительские элементы, взятые из отображения и вставить ее в качестве внутреннего HTML-содержимого во вновь созданный элемент разметки

В данном примере кода создается отображение для всех типов элементов разметки, которые требуется разместить в специальных родительских контейнерах. В этом отображении указано количество дескрипторов в новом узле, а также код открывающих и закрывающих родительских дескрипторов. Затем в данном примере кода применяется регулярное выражение для сопоставления с открывающей угловой скобкой и именем дескриптора того элемента разметки, который требуется вставить, как показано ниже.

```
const tagName = htmlString.match(/<\w+/);
```

Далее в рассматриваемом здесь примере кода выбирается элемент отображения. И если такой элемент отсутствует, то создается фиктивный элемент с пустым родительским элементом разметки:

```
let mapEntry = tagName ? map[tagName[0]] : null;
if (!mapEntry) { mapEntry = [0, " ", " "]; }
```

После этого создается новый элемент разметки `div`, в который будет помещен отображаемый элемент HTML-разметки вместе с кодом своего родительского контейнера:

```
let div = (doc || document).createElement("div");
div.innerHTML = mapEntry[1] + htmlString + mapEntry[2]
```

И, наконец, в данном примере кода выполняется поиск родительского элемента требуемого узла, создаваемого из исходной HTML-строки, после чего возвращается вновь созданный узел, как показано ниже.

```
while (mapEntry[0]--) { div = div.lastChild;}
return div.childNodes;
```

В результате получится ряд узлов DOM, которые уже можно вставить в HTML-документ.

Если вернуться к основному назначению данного примера и применить функцию `getNodes()`, то в конечном итоге получится HTML-код, аналогичный приведенному ниже.

```
<select multiple>
  <option>Yoshi</option>
  <option>Kuma</option>
</select>
<table></table>
```

Элементы разметки `option`, заключенные в оболочку элемента разметки `select`

12.1.2. Вставка элементов разметки в документ

Сформировав конкретные узлы модели DOM, можно перейти непосредственно к их вставке в документ. Для этого придется предпринять ряд шагов, рассматриваемых в этом разделе.

У нас уже имеется массив элементов, которые требуется вставить в произвольных местах документа, поэтому мы можем попытаться сократить число операций, необходимых для этого. Сделать это можно с помощью *фрагментов* DOM – части спецификации модели DOM по стандарту W3C, поддерживаемой во всех браузерах. Это полезное средство предоставляет контейнер для хранения целой коллекции узлов DOM.

Помимо того, что такая возможность полезна сама по себе, она дает следующие преимущества: фрагмент можно вставлять и клонировать в единой операции, а не вставлять и клонировать каждый узел по отдельности. Благодаря этому значительно сокращается количество операций, которые требуется выполнить на странице.

Прежде чем воспользоваться этим механизмом в коде, вернемся к исходному коду функции `getNodes()` из листинга 12.2 и немного отредактируем его, чтобы применить в нем фрагменты DOM. Эти изменения минимальны и состоят в добавлении параметра `fragment` в список параметров данной функции, как показано в примере кода из листинга 12.3.

Листинг 12.3. Расширение функции `getNodes()` фрагментами DOM

```
function getNodes(htmlString, doc, fragment){
  const map = {
    "<td":[3,"<table><tbody><tr>","</tr></tbody></table>"],
    "<th":[3,"<table><tbody><tr>","</tr></tbody></table>"],
    "<tr":[2,"<table><thead>","</thead></table>"],
    "<option":[1,"<select multiple>","</select>"],
    "<optgroup":[1,"<select multiple>","</select>"],
    "<legend":[1,"<fieldset>","</fieldset>"],
    "<thead":[1,"<table>","</table>"],
    "<tbody":[1,"<table>","</table>"],
    "<tfoot":[1,"<table>","</table>"],
    "<colgroup":[1,"<table>","</table>"],
    "<caption":[1,"<table>","</table>"],
    "<col":[2,"<table><tbody></tbody><colgroup>","</colgroup></table>"],
  };
  const tagName = htmlString.match(/<\w+/);
  let mapEntry = tagName ? map[tagName[0]] : null;
  if (!mapEntry) { mapEntry = [0, " "," " ];}
```

Добавить новый параметр `fragment` в определение функции

```
  let div = (doc || document).createElement("div");
  div.innerHTML = mapEntry[1] + htmlString + mapEntry[2];
  while (mapEntry[0]--) { div = div.lastChild;}
  if (fragment) {
    while (div.firstChild) {
      fragment.appendChild(div.firstChild);
    }
  }

  return div.childNodes;
}
```

Если фрагмент существует, вставить в него узлы

В исходный код из листинга 12.3 внесен ряд изменений. Прежде всего, изменена сигнатура функции, где введен еще один параметр `fragment`:

```
function getNodes(htmlString, doc, fragment) {...}
```

Если этот параметр передается данной функции, то предполагается, что им должен быть фрагмент DOM, в который требуется вставить узлы для последующего применения. Чтобы вставить узлы в передаваемый фрагмент модели DOM, достаточно добавить следующий фрагмент кода непосредственно перед оператором `return`:

```
if (fragment) {
  while (div.firstChild) {
    fragment.appendChild(div.firstChild);
  }
}
```

А теперь применим рассматриваемый здесь механизм на практике. В примере кода, приведенном в листинге 12.4, предполагается, что обновленная функция `getNodes()` находится в области видимости, а фрагмент создается и передается этой функции для преобразования входящей HTML-строки в элементы модели DOM. И эти элементы автоматически добавляются ко вновь созданному фрагменту.

Листинг 12.4. Вставка фрагмента в нескольких местах модели DOM

```
<div id="test"><b>Hello</b>, I'm a ninja!</div>
<div id="test2"></div>
<script>
  document.addEventListener("DOMContentLoaded", () => {
    function insert(elems, args, callback) {
      if (elems.length) {
        const doc = elems[0].ownerDocument || elems[0],
              fragment = doc.createDocumentFragment(),
              scripts = getNodes(args, doc, fragment),
              first = fragment.firstChild;
        if (first) {
          for (let i = 0; elems[i]; i++) {
            callback.call(root(elems[i], first),
```

Создать пару тестовых узлов

Создать фрагмент документа для последующей вставки узлов

Создать HTML-узлы из HTML-строки

```
            i > 0 ? fragment.cloneNode(true) : fragment);
          }
        }
      }
    }
    const divs = document.querySelectorAll("div");
    insert(divs, "<b>Name:</b>", function (fragment) {
      this.appendChild(fragment);
    });

    insert(divs, "<span>First</span> <span>Last</span>",
      function (fragment) {
        this.parentNode.insertBefore(fragment, this);
      });
  });
</script>
```

Если требуется вставить узлы не только в один элемент, то каждый раз придется клонировать фрагмент

Необходимо также обратить внимание на следующее: если данный элемент разметки вставляется в нескольких местах документа, то придется неоднократно клонировать соответствующий фрагмент. И если бы не было фрагментов, то нам пришлось бы всякий раз клонировать каждый узел в отдельности, а не весь фрагмент сразу.

Выше мы рассмотрели механизм формирования и вставки произвольных элементов модели DOM интуитивным способом. Продолжим исследование модели DOM, чтобы выяснить, чем атрибуты модели DOM отличаются от ее свойств.

12.2. Атрибуты и свойства модели DOM

Доступ к значениям атрибутов элементов разметки можно получить двумя путями: используя традиционные для модели DOM методы `getAttribute()` и `setAttribute()` или же свойства самих объектов DOM, имена которых соответствуют атрибутам. Чтобы, например, получить значение атрибута `id` элемента разметки, ссылка на который хранится в переменной `e`, можно воспользоваться одним из следующих способов:

```
e.getAttribute('id')
e.id
```

В любом случае будет получено значение атрибута `id`.

Чтобы стало понятнее, каким образом ведут себя значения атрибутов и соответствующих им свойств, обратимся к примеру кода, приведенного в листинге 12.5.

Листинг 12.5. Доступ к значениям атрибутов через методы и свойства модели DOM

```
<div></div>
<script>
  document.addEventListener("DOMContentLoaded", () => {
    const div = document.querySelector("div");
```

Получить ссылку на элемент

```
    div.setAttribute("id","ninja-1");
    assert(div.getAttribute('id') === "ninja-1",
           "Attribute successfully changed");
    div.id = "ninja-2";
    assert(div.id === "ninja-2",
           "Property successfully changed");
    assert(div.getAttribute('id') === "ninja-2",
           "Attribute successfully changed via property");
    div.setAttribute("id","ninja-3");
    assert(div.id === "ninja-3",
           "Property successfully changed via attribute");
    assert(div.getAttribute('id') === "ninja-3",
           "Attribute successfully changed");
  });
</script>
```

Изменить значение атрибута id с помощью метода setAttribute() и проверить, изменилось ли это значение

Изменить значение свойства и проверить, изменилось ли это значение

Используя метод setAttribute(), изменить также значение, полученное из свойства

При изменении свойства изменяется также значение, получаемое с помощью метода getAttribute()

В данном примере кода демонстрируется любопытное поведение в отношении свойств и атрибутов элементов разметки. Сначала в нем определяется простой элемент разметки , используемый далее в качестве субъекта тестирования. Чтобы убедиться, что построение модели DOM завершено, в обработчике событий DOMContentLoaded(), вызываемом при загрузке документа, сначала получается ссылка на единственный элемент разметки , т.е. const div = document.querySelector("div"), а затем выполняется ряд тестов.

В первом тесте значение "ninja-1" атрибута id устанавливается с помощью метода setAttribute(). Затем утверждается, что метод getAttribute() возвращает то же самое значение данного атрибута. И не должно быть ничего удивительного в том, что этот тест пройдет при загрузке веб-страницы, как показано ниже.

```
div.setAttribute("id", "ninja-1");
assert(div.getAttribute('id') === "ninja-1",
       "Attribute successfully changed");
```

Аналогично в следующем тесте сначала устанавливается значение "ninja-2" свойства id, а затем проверяется, действительно ли это значение изменилось предполагаемым образом:

```
div.id = "ninja-2";
assert(div.id === "ninja-2",
       "Property successfully changed");
```

В следующем тесте уже происходит нечто любопытное. Здесь снова устанавливается новое значение свойства id (на этот раз "ninja-3") и проверяется, изменилось ли значение этого свойства. Но затем утверждается также, что измениться должно не только значение свойства, но и значение *атрибута* id. И оба утверждения проходят. Из этого теста можно сделать следующий вывод: свойство id и атрибут id каким-то образом связаны вместе. При изменении значения свойства id изменяется также значение атрибута id.

```
div.id = "ninja-3";
assert(div.id === "ninja-3",
       "Property successfully changed");
assert(div.getAttribute('id') === "ninja-3",
       "Attribute successfully changed via property");
```

А в следующем тесте этот факт подтверждается иначе: при установке значения атрибута изменяется также значение соответствующего свойства:

```
div.setAttribute("id","ninja-4");
assert(div.id === "ninja-4",
       "Property successfully changed via attribute");
assert(div.getAttribute('id') === "ninja-4",
       "Attribute changed");
```

Но сделанный выше тест не должен вводить вас в заблуждение относительно того, что свойство и атрибут имеют общее значение, поскольку это на самом деле не так. Как будет показано далее в этой главе, несмотря на существующую явную связь между атрибутом и свойством, их значения далеко не всегда одинаковы.

Следует особо подчеркнуть, что не для всех атрибутов существуют соответствующие им свойства элементов разметки. И хотя это справедливо в целом для атрибутов, исходно указываемых в модели DOM, для *специальных атрибутов*, которые могут размещаться в элементах HTML-разметки веб-страниц, соответствующие свойства не создаются. Поэтому для доступа к значению специального атрибута придется вызывать методы `getAttribute()` и `setAttribute()` модели DOM.

Если неизвестно, существует ли свойство для конкретного атрибута, его наличие можно всегда проверить. И если оно не существует, то в качестве резервного варианта можно всегда обратиться к методам модели DOM. В приведенном ниже примере показано, как это реализуется непосредственно в коде.

```
const value = element.someValue ? element.someValue
                                 : element.getAttribute('someValue');
```

Совет

Имена всех специальных атрибутов должны начинаться с префикса **data-**, чтобы обеспечить их корректность согласно спецификации HTML5. Помимо прочего, это удобное условное обозначение имен, четко отделяющее специальные атрибуты от встроенных.

12.3. Трудности обращения с атрибутами стилевого оформления

Аналогично общим атрибутам, получение доступа и установка атрибутов стилевого оформления может вызвать серьезные трудности. Как и для обращения с атрибутами и свойствами, рассмотренными в предыдущем разделе,

в данном случае имеются два подхода: обращаться непосредственно к значению атрибута `style` или же к образуемому из него свойству элемента разметки.

Из этих двух подходов чаще всего применяется обращение к свойству `style` элемента разметки, которое содержит не символьную строку, а объект со свойствами, соответствующими значениям стилевого оформления, указанным в разметке элемента. Помимо этого, имеется метод для доступа к текущей информации о вычисленном стилевом оформлении элемента, где *вычисленное стилевое оформление*, по существу, означает конкретные стили, которые будут применяться к элементу после анализа информации о всех наследуемых и применяемых стилях.

В этом разделе поясняется, каким образом стилевое оформление интерпретируется в браузерах. И начнем мы с выяснения места, где указывается информация о стилевом оформлении.

12.3.1. Местонахождение стилей

Информация о стилевом оформлении, находящаяся в свойстве `style` элемента модели DOM, первоначально устанавливается, исходя из значения, указанного в атрибуте `style` разметки данного элемента. Например, в результате операции присваивания `style="color:red;"` информация о стилевом оформлении размещается в объекте `style`. А во время выполнения сценария на веб-странице значения свойств могут быть установлены или видоизменены в объекте `style`, причем эти изменения будут активно воздействовать на внешний вид элемента.

Многие авторы сценариев с разочарованием обнаруживают, что ни одно из значений элементов `<style>` разметки страницы или внешних таблиц стилей недоступно в объекте `style` данного элемента. Но настоящим мастерам не пристало пребывать долго в разочаровании, поэтому мы попытаемся далее найти способ получить подобную информацию. А до тех пор выясним, каким образом свойство `style` получает свои значения, обратившись к примеру кода, из листинга 12.6.

Листинг 12.6. Исследование свойства `style`

Объявить таблицу стилей, расположенную на странице, с информацией о размере шрифта и виде рамки

Этому тестируемому элементу должны быть назначены несколько стилей из разных мест, включая его собственный атрибут `style` и таблицу стилей

Проверить, зарегистрирован ли встраиваемый стиль цвета

```
<style>
  div { font-size: 1.8em; border: 0 solid gold; }
</style>
<div style="color:#000;" title="Ninja power!">
  忍者パワー
</div>
<script>
  document.addEventListener("DOMContentLoaded", () => {
    const div = document.querySelector("div");
    assert(div.style.color === 'rgb(0, 0, 0)' ||
           div.style.color === '#000',
```

```
        'color was recorded');
    assert(div.style.fontSize === '1.8em',
        'fontSize was recorded');
    assert(div.style.borderWidth === '0',
        'borderWidth was recorded');
    div.style.borderWidth = "4px";
    assert(div.style.borderWidth === '4px',
        'borderWidth was replaced');
  });
</script>
```

Проверить, зарегистрирован ли наследуемый стиль размера шрифта

Проверить, зарегистрирован ли наследуемый стиль ширины рамки

Заменить стиль ширины рамки

Проверить, зарегистрировано ли изменение стиля ширины рамки

В данном примере кода сначала создается элемент разметки `<style>` для задания внутренней таблицы стилей, значения в которой будут далее применяться к элементам разметки веб-страницы. В этой таблице стилей определено, что текст во всех элементах разметки `<div>` должен быть набран шрифтом, размер которого в 1,8 раза крупнее выбираемого по умолчанию, и обрамлен сплошной золотистой рамкой нулевой толщины. Это означает, что любые элементы, к которым применяется такое стилевое оформление, будут иметь невидимую рамку из-за ее нулевой толщины.

```
<style>
  div { font-size: 1.8em; border: 0 solid gold; }
</style>
```

Затем создается элемент разметки **`<div>`** со встроенным стилевым атрибутом, определяющим черную окраску текста в данном элементе разметки:

```
<div style="color:#000;" title="Ninja power!">
  忍者パワー
</div>
```

Далее начинается тестирование. После получения ссылки на элемент разметки `<div>` проверяется, приобрел ли его атрибут `style` свойство `color`, представляющее цвет, присвоенный данному элементу, как показано ниже. Следует иметь в виду, что в большинстве браузеров свойство `color` нормализуется в формате RGB, когда его значение устанавливается в свойстве `style`, несмотря на то, что во встраиваемом стиле оно указано в виде **`#000`**. Именно поэтому оно и проверяется в обоих форматах. Как показано на рис. 12.1, данный тест проходит.

```
assert(div.style.color === 'rgb(0, 0, 0)' ||
       div.style.color === '#000',
       'color was recorded');
```

Далее проверяется, зарегистрированы ли в объекте `style` размер шрифта и толщина рамки, указанные в таблице встраиваемых стилей, как показано ниже. И хотя на рис. 12.1 показано, что стиль, определяющий размер шрифта, применен к элементу разметки, его тест все равно не проходит. Дело в том, что содержимое объекта `style` не отражает информацию о стилевом оформлении, наследуемую из вложенных таблиц стилей (CSS).

```
assert(div.style.fontSize === '1.8em',
       'fontSize was recorded');
assert(div.style.borderWidth === '0',
       'borderWidth was recorded');
```

Рис. 12.1. Как показывают результаты тестирования, встраиваемые и присваиваемые стили регистрируются, тогда как наследуемые — нет

После этого значение свойства `borderWidth` в объекте `style` устанавливается равным **4** пикселям, а далее проверяется, было ли применено это изменение, как показано ниже. И как следует из рис. 12.1, тест данного изменения проходит, и невидимая ранее рамка теперь окружает элемент разметки страницы. В результате такого присваивания значение свойства `borderWidth` оказывается в свойстве `style` данного элемента разметки, что и подтверждает тест.

```
div.style.borderWidth = "4px";
assert(div.style.borderWidth === '4px',
       'borderWidth was replaced');
```

Следует иметь в виду, что любые значения, устанавливаемые в свойстве `style` элемента разметки, будут получать приоритет над всем, что наследуется из таблицы стилей, даже если в элементе таблицы стилей используется аннотация `!important` (важно!).

Возможно, вы обратили внимание на то, что в коде из листинга 12.6 свойство размера шрифта обозначается в таблице CSS как `font-size`, а в сценарии — как `fontSize`. Чем объясняется такое расхождение? Ответ на этот вопрос мы постараемся найти в следующем разделе.

12.3.2. Именование свойств стилевого оформления

С атрибутами стилевого оформления связано относительно немного затруднений кросс-браузерного характера, если речь идет о доступе к значениям, предоставляемым браузером. Тем не менее существуют отличия в именовании стилей в таблицах CSS, сценариях, а также в браузерах.

Имена атрибутов стилевого оформления в таблицах CSS, составленные из нескольких слов, обычно разделяются дефисом, например `font-weight`, `font-size` и `background-color`. Напомним, что в JavaScript имена свойств также *могут* разделяться дефисом, но в таком случае они становятся недоступными через операцию-точку. Например, приведенная ниже строка кода считается вполне допустимой.

```
const fontSize = element.style['font-size'];
```

А следующая строка кода считается недопустимой:

```
const fontSize = element.style.font-size;
```

Синтаксический анализатор JavaScript интерпретирует дефис в этой строке кода как операцию вычитания, что, вероятнее всего, приведет к малоутешительным последствиям. Поэтому многословные имена стилей в таблицах CSS обычно приводятся к смешанному ("верблюжьему") написанию, когда они употребляются в качестве имен свойств, чтобы не вынуждать разработчиков веб-страниц пользоваться только общей формой доступа к свойствам. Таким образом, имя `font-size` преобразуется в `fontSize`, имя `background-color` — в `backgroundColor`. Чтобы не помнить об этом постоянно, можно создать простую функцию, в которой стили устанавливаются и получаются с автоматическим приведением к форме смешанной записи, как демонстрируется в примере кода из листинга 12.7.

Листинг 12.7. Простой способ организации доступа к стилям

```
<div style="color:red;font-size:10px;background-color:#eee;"></div>
<script>
  function style(element,name,value){
    name = name.replace(/-([a-z])/ig, (all,letter) => {
      return letter.toUpperCase();
    });
    if (typeof value !== 'undefined') {
      element.style[name] = value;
    }
    return element.style[name];
  }
  document.addEventListener("DOMContentLoaded", () => {
    const div = document.querySelector("div");
    assert(style(div,'color') === "red", style(div,'color'));
    assert(style(div,'font-size') ===
           "10px", style(div,'font-size'));
    assert(style(div,'background-color') ===
           "rgb(238, 238, 238)",style(div,'background-color'));
  });
</script>
```

Определить функцию обращения к стилям, где свойству стиля присваивается значение, если оно предоставляется, а иначе — возвращается текущее значение свойства. Этой функцией можно пользоваться как для установки, так и для получения значения свойства

Привести имя к смешанному написанию

Новое значение свойства стиля устанавливается, если оно предоставляется

Всегда возвращать значение свойства стиля

Функция обращения к стилям должна обладать следующими характеристиками.

- В ней применяется регулярное выражение для преобразования значения параметра `name` в смешанное написание (если применение регулярных выражений вызывает у вас трудности, обратитесь к главе 10).
- Ею можно пользоваться как для установки, так и для получения значений свойств стилей, в зависимости от указанного списка аргументов. Например, сделав вызов `style(div, 'font-size')`, можно получить значение свойства `font-size`, а в результате вызова `style(div, 'font-size', '5px')` — установить новое значение в этом свойстве.

В качестве примера рассмотрим следующий фрагмент кода:

```
function style(element,name,value){
    ...
  if (typeof value !== 'undefined') {
    element.style[name] = value;
  }

  return element.style[name];
}
```

Если данной функции передается значение аргумента `value`, она присваивает переданное ей значение атрибута указанному свойству. А если аргумент `value` опускается, то данная функция возвращает значение указанного атрибута. Но в любом случае возвращается значение атрибута, и благодаря этому упрощается применение данной функции в цепочке вызовов как для установки, так и для получения значений свойств и атрибутов.

Свойство `style` элемента разметки не содержит никакой информации о стилевом оформлении, наследуемой этим элементом из доступных ему таблиц стилей. Но зачастую полезно знать полностью вычисленный стиль, применяемый к элементу. Поэтому выясним, каким образом можно получить его.

12.3.3. Извлечение вычисленных стилей

В любой момент времени *вычисленный стиль* элемента разметки представляет собой определенное сочетание всех встроенных стилей, предоставляемых браузером, всех стилей, применяемых к данному элементу с помощью таблиц стилей, атрибута `style` и любых манипуляций со свойством `style` в сценарии. На рис. 12.2 наглядно показано, каким образом инструментальные средства разработки в браузере различают стили.

Для получения вычисленных стилей во всех современных браузерах реализован метод `getComputedStyle()`. Ему передается элемент, стили которого должны быть вычислены, а он возвращает интерфейс, посредством которого можно делать запросы к свойствам данного элемента. Так, для выборки вычисленного стиля конкретного свойства стилевого оформления в возвращаемом интерфейсе предоставляется метод `getPropertyValue()`. В отличие от

свойств объекта `style` элемента разметки, методу `getPropertyValue()` передаются имена свойств стилевого оформления в формате CSS (например, `font-size` и `background-color`), а не в смешанном написании. Простой пример извлечения вычисленного стиля демонстрируется в коде из листинга 12.8.

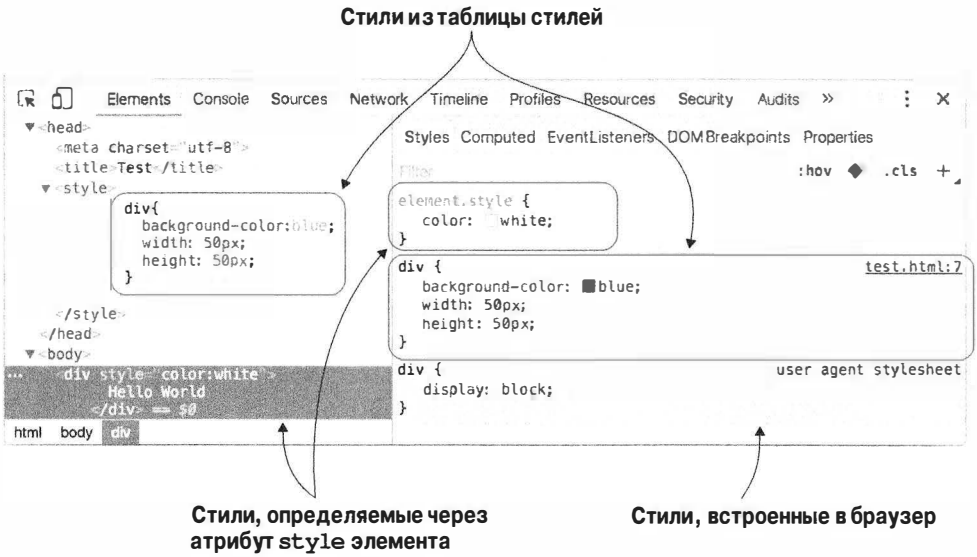


Рис. 12.2. Окончательный стиль оформления элемента может зависеть от многих факторов: встроенных в браузер стилей (из таблицы стилей пользовательского агента), стилей, назначаемых через свойство **`style`**, а также стилей, устанавливаемых правилами, определяемыми в коде CSS

Листинг 12.8. Извлечение вычисленных стилей

Определить функцию для получения вычисленного значения свойства `style`

```
<style>                                          Создать субъект тестирования с атрибутом style
  div {
    background-color: #ffc; display: inline; font-size: 1.8em;
    border: 1px solid crimson; color: green;
  }
</style>
<div style="color:crimson;" id="testSubject" title="Ninja power!">
  忍者パワー
</div>
<script>                                         Вызвать встроенный метод getComputedStyle(), чтобы получить объект дескриптора
  function fetchComputedStyle(element,property) {
    const computedStyles = getComputedStyle(element);
    if (computedStyles) {                        Заменить смешанное написание написанием через дефис
      property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
```

```
        return computedStyles.getPropertyValue(property);
      }
    }
    document.addEventListener("DOMContentLoaded", () => {
      const div = document.querySelector("div");
      report("background-color: " +
             fetchComputedStyle(div,'background-color'));
      report("display: " +
             fetchComputedStyle(div,'display'));
      report("font-size: " +
             fetchComputedStyle(div,'fontSize'));
      report("color: " +
             fetchComputedStyle(div,'color'));
      report("border-top-color: " +
             fetchComputedStyle(div,'borderTopColor'));
      report("border-top-width: " +
             fetchComputedStyle(div,'border-top-width'));
    });
  </script>
```

Проверить, можно ли получить значения различных свойств, используя разные написания

Чтобы проверить вновь созданную функцию, в приведенном выше примере кода создается элемент разметки, в котором указывается информация о его стилевом оформлении, а также внешняя таблица стилей, предоставляющая правила стилевого оформления, которые должны применяться к данному элементу. При этом предполагается, что вычисленные стили должны отражать результат как непосредственного, так и внешнего стилевого оформления данного элемента.

Затем определяется новая функция, которой в качестве аргументов передаются: элемент разметки и свойство стилевого оформления, для которого требуется найти вычисленное значение. Ради особого удобства многословные имена свойств могут указываться в одном из двух форматов: через дефис или в смешанном написании. Иными словами, данная функция должна распознавать свойство стилевого оформления как под именем `backgroundColor`, так и под именем `background-color`. Ниже поясняется, как этого добиться.

Прежде всего требуется получить интерфейс для вычисленного стиля. Этот интерфейс сохраняется в переменной `computedStyles` для последующего обращения, как показано ниже. И это требуется сделать потому, что заранее неизвестно, насколько затратной может оказаться подобная операция, а следовательно, было бы желательно избегать ее повторения без особой надобности.

```
const computedStyles = getComputedStyle(element);
if (computedStyles) {
  property = property.replace(/([A-Z])/g,'-$1').toLowerCase();
  return computedStyles.getPropertyValue(property);
}
```

Если интерфейс будет получен успешно, а особых оснований предположить иной исход не существует, хотя и бдительность терять нельзя, то из данного интерфейса вызывается метод `getPropertyValue()` с целью получить значение

вычисленного стиля. Но прежде необходимо привести имя свойства стилевого оформления к написанию через дефис или к смешанному написанию. Так, в методе `getPropertyValue()` предполагается получить имя в написании через дефис, и поэтому для вставки дефиса перед каждой прописной буквой в имени свойства и преобразования всего имени в строчные буквы вызывается метод `replace()` из класса `String` с простым, но искусным регулярным выражением.

Для проверки вновь созданной функции делается целый ряд ее вызовов, в которых ей передаются различные имена стилей в разных форматах, а результаты тестирования выводятся на экран, как показано на рис. 12.3.

Рис. 12.3. Вычисленные стили включают в себя все стили, указанные в элементе разметки, а также те, что наследуются из таблиц стилей

Следует иметь в виду, что стили извлекаются независимо от того, были ли они явным образом объявлены в элементе разметки или унаследованы из таблицы стилей. Кроме того, значение свойства `color`, указанного как в таблице стилей, так и непосредственно в элементе разметки, возвращается в виде явно заданного значения. Стили, задаваемые в атрибуте `style` элемента разметки, всегда получают приоритет над наследуемыми стилями, даже если они и отмечены аннотацией `!important` в таблице стилей.

Обращаясь со свойствами стилевого оформления, следует также иметь в виду *сочетание* свойств. В таблицах CSS допускается краткая запись сочетания свойств, например, свойств рамки. Вместо того чтобы указывать цвета, ширину и стили обрамления по отдельности для всех четырех рамок, можно записать следующее правило их стилевого оформления:

```
border: 1px solid crimson;
```

Именно это правило и было использовано в листинге 12.8. Оно позволяет сэкономить на наборе текста стилей, но не следует забывать, что стили нужно извлекать из отдельных свойств более низкого уровня. В частности, нельзя извлечь стиль `border`, но можно извлечь стили `border-top-color` и `border-`

top-width, что и было сделано в приведенном выше примере. Такой способ может доставить немного хлопот, особенно если всем четырем стилям присвоены одинаковые значения. Впрочем, и это препятствие нетрудно обойти, как пояснялось ранее.

12.3.4. Преобразование значений, указываемых в пикселях

При установке значений атрибутов стилевого оформления следует обращать особое внимание на присваивание числовых значений, указываемых в пикселях. Устанавливая числовое значение в свойстве стилевого оформления, необходимо указывать одинаковые единицы измерения для надежной работы прикладного кода во всех браузерах. Допустим, требуется установить значение 10 пикселей в свойстве height стилевого оформления элемента разметки вебстраницы. Для этой цели можно воспользоваться одним из следующих двух способов, обеспечивающих надежную работу прикладного кода во всех браузерах:

```
element.style.height = "10px";
element.style.height = 10 + "px";
```

А приведенный ниже способ нельзя считать надежным.

```
element.style.height = 10;
```

Казалось бы, достаточно добавить немного логики в функцию style() из листинга 12.7, чтобы присоединить обозначение единиц измерения "px" к числовому значению, передаваемому данной функции, но не все так просто! Далеко не все числовые значения представлены в пикселях! Существует целый ряд свойств стилевого оформления, числовые значения которых заданы в других единицах измерения. К их числу относятся следующие свойства:

- z-index
- font-weight
- opacity
- zoom
- line-height

Для указания числовых значений, присваиваемых этим (и любым другим мыслимым) свойствам, можно соответственно расширить функцию style() из листинга 12.6, чтобы автоматически интерпретировать значения, указываемые в других единицах измерения, кроме пикселей.

Кроме того, для чтения значений, указываемых в пикселях, из атрибута стилевого оформления следует пользоваться методом parseFloat(), чтобы в любом случае получить предполагаемое значение. А теперь перейдем к рассмотрению ряда важных свойств стилевого оформления, обращение с которыми может вызвать определенные трудности.

12.3.5. Указание размеров по высоте и по ширине

Обращение со свойствами стилевого оформления вроде `height` и `width` вызывает особые трудности, поскольку им назначается стандартное значение `auto`, если оно не указано специально. Это означает, что размеры элемента разметки веб-страницы устанавливаются автоматически, исходя из его содержимого. В итоге свойствами `height` и `width` нельзя воспользоваться для получения точных значений, если только не указать их значения явным образом в символьной строке соответствующего атрибута стилевого оформления.

Правда, свойства `offsetHeight` и `offsetWidth` обеспечивают довольно надежный способ доступа к правильному значению высоты и ширины элемента. Но не следует забывать, что значения, присваиваемые этим двум свойствам, включают в себя поля вокруг элемента разметки. Такая информация зачастую весьма желательна, если требуется расположить один элемент над другим. Но иногда информацию о размерах элементов требуется получить как с указанием ширины рамок и полей, так и без них.

Следует, однако, иметь в виду, что на веб-сайтах с повышенной интерактивностью элементы разметки веб-страниц могут находиться некоторое время в неотображаемом состоянии, когда значение свойства `display` стилевого оформления установлено равным `none`. А когда такой элемент не отображается, у него отсутствуют размеры. И в результате любой попытки извлечь значения свойств `offsetWidth` и `offsetHeight` неотображаемого элемента разметки будет получено нулевое значение.

Если требуется получить нескрытые размеры элементов, скрываемых подобным образом, то можно воспользоваться специальным приемом, чтобы раскрыть элемент разметки на мгновение, извлечь из него значение, а затем скрыть его снова. Разумеется, это нужно сделать совершенно незаметно, выполнив следующую последовательность действий.

1. Изменить значение свойства `display` на `block`.
2. Установить значение `hidden` свойства `visibility`.
3. Установить значение `absolute` свойства `position`.
4. Извлечь значения размеров.
5. Восстановить исходные значения упомянутых выше свойств.

Благодаря изменению значения свойства `display` на `block` появляется возможность извлечь конкретные значения свойств `offsetHeight` и `offsetWidth`, но в таком случае элемент разметки включается в отображаемую часть веб-страницы, а следовательно, он становится видимым. А для того чтобы сделать его невидимым, устанавливается значение `hidden` свойства `visibility`. Но здесь, как всегда, возникает следующая загвоздка: на месте неотображаемого элемента разметки веб-страницы появляется крупная прореха. Поэтому свойству `position` присваивается значение `absolute`, чтобы исключить его из обычной последовательности отображения содержимого веб-страницы.

Реализовать такой прием намного проще, чем описать его суть, что и демонстрируется в примере кода из листинга 12.9.

Листинг 12.9. Извлечение размеров скрытых элементов разметки

```
<div>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Suspendisse congue facilisis dignissim. Fusce sodales,
  odio commodo accumsan commodo, lacus odio aliquet purus,
  <img src="../images/ninja-with-pole.png" id="withPole" alt="ninja pole"/>
  <img src="../images/ninja-with-shuriken.png"
    id="withShuriken" style="display:none" alt="ninja shuriken" />
  vel rhoncus elit sem quis libero. Cum sociis natoque
  penatibus et magnis dis parturient montes, nascetur
  ridiculus mus. In hac habitasse platea dictumst. Donec
  adipiscing urna ut nibh vestibulum vitae mattis leo
  rutrum. Etiam a lectus ut nunc mattis laoreet at
  placerat nulla. Aenean tincidunt lorem eu dolor commodo
  ornare.
</div>
<script
  (function(){                                    Создать локальную область видимости
    const PROPERTIES = {                          Определить целевые свойства
      position: "absolute",
      visibility: "hidden",
      display: "block"
    };
    window.getDimensions = element => {           Создать новую функцию
      const previous = {};                        Запомнить настройки
      for (let key in PROPERTIES) {
        previous[key] = element.style[key];
        element.style[key] = PROPERTIES[key];     Заменить настройки
      }
      const result = {                            Извлечь размеры
        width: element.offsetWidth,
        height: element.offsetHeight
      };
      for (let in PROPERTIES) {                   Восставить настройки
        element.style[key] = previous[key];
      }
      return result;
    };
  })();
  document.addEventListener("DOMContentLoaded", () => {
    setTimeout(() => {
      const withPole = document.getElementById('withPole'),
          withShuriken = document.getElementById('withShuriken');
      assert(withPole.offsetWidth === 41,
             "Pole image width fetched; actual: " +
             withPole.offsetWidth + ", expected: 41");   Проверить видимый элемент
      assert(withPole.offsetHeight === 48,
```

```
            "Pole image height fetched: actual: " +
            withPole.offsetHeight + ", expected 48");
      assert(withShuriken.offsetWidth === 36,
            "Shuriken image width fetched; actual: " +
            withShuriken.offsetWidth + ", expected: 36");
      assert(withShuriken.offsetHeight === 48,
            "Shuriken image height fetched: actual: " +
            withShuriken.offsetHeight + ", expected 48");
      const dimensions = getDimensions(withShuriken);
      assert(dimensions.width === 36,
            "Shuriken image width fetched; actual: " +
            dimensions.width + ", expected: 36");
      assert(dimensions.height === 48,
            "Shuriken image height fetched: actual: " +
            dimensions.height + ", expected 48");
    },3000);
  });
</script>
```

Проверить невидимый элемент

Использовать новую функцию

Еще раз проверить невидимый элемент

Приведенный выше листинг довольно длинный, но бо́льшую его часть занимает тестовый код, тогда как реализация новой функции для извлечения размеров элемента разметки — лишь около десятка строк кода. Рассмотрим данный пример кода по частям. Сначала в нем создается ряд элементов разметки для последующего тестирования. В частности, элемент разметки `<div>` содержит фрагмент текста, заверстанный вокруг двух изображений и выровненный по левому краю, для чего используются стили из внешней таблицы стилей. А элементы разметки обоих изображений устанавливаются в качестве объектов для последующего тестирования. С этой целью один из них сделан видимым, а другой — невидимым.

Перед выполнением сценария элементы разметки веб-страницы выглядят так, как показано на рис. 12.4. Если не скрыть второе изображение ниндзя, оно появится рядом с первым.

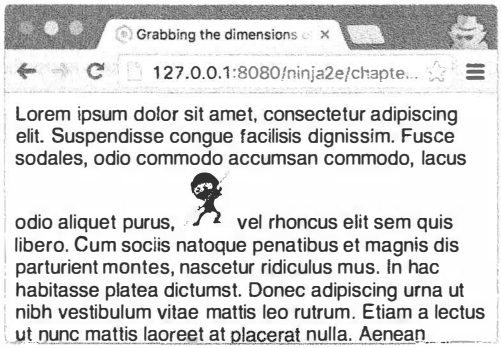


Рис. 12.4. Для тестирования рассматриваемого здесь приема выборки размеров элементов используются два изображения: одно — видимое, другое — скрытое

Затем определяется новая функция. Для хранения важной информации предполагается использовать ассоциативный массив, но чтобы не засорять глобальное пространство имен, данная функция должна быть доступна только в своей локальной области действия.

С этой целью определение ассоциативного массива и объявление функции заключается в тело немедленно вызываемой функции, образующей локальную область видимости. Ассоциативный массив недоступен за пределами немедленно вызываемой функции, но функция `getDimensions()`, также определяемая в теле немедленно вызываемой функции, получает доступ к ассоциативному массиву через замыкание, как показано ниже.

```
function(){
  const PROPERTIES = {
    position: "absolute",
    visibility: "hidden",
    display: "block"
  };
  window.getDimensions = element => {
    const previous = {};
    for (let key in PROPERTIES) {
      previous[key] = element.style[key];
      element.style[key] = PROPERTIES[key];
    }
    const result = {
      width: element.offsetWidth,
      height: element.offsetHeight
    };
    for (let key in PROPERTIES) {
      element.style[key] = previous[key];
    }
    return result;
  };
})();
```

Далее объявляется новая функция для выборки размеров элемента разметки. В качестве аргумента этой функции передается элемент разметки, размеры которого требуется выбрать для последующей обработки. В теле этой функции сначала создается хеш-массив `previous`, в котором будут храниться предыдущие значения свойств стилевого оформления, чтобы их можно было заменить, а затем и восстановить. С этой целью организуется циклическое обращение к каждому предыдущему свойству и их замена новыми значениями.

Сделав все это, можно далее перейти к выборке размеров элемента разметки, который теперь невидим и занимает абсолютное положение в отображаемой верстке веб-страницы. Выбранные размеры этого элемента сохраняются в хеш-массиве, присвоенном локальной переменной `result`.

После извлечения нужной информации следы этой скрытной операции тщательно заметаются путем восстановления исходных значений свойств стилевого оформления, в результате чего возвращается ассоциативный массив,

содержащий свойства `width` и `height`. Все это выглядит красиво в теории, а как оно будет работать на практике? Попробуем найти ответ и на этот вопрос.

В обработчике событий, связанных с загрузкой веб-страницы, тесты выполняются в функции обратного вызова таймера, срабатывающего через 3 секунды. А зачем это нужно? Подобным образом в обработчике событий гарантируется, что тест не будет выполняться до тех пор, пока не станет известно, что модель DOM построена. А таймер позволяет наблюдать отображаемое содержимое веб-страницы по ходу выполнения теста, исключая сбои в отображении на время манипулирования свойствами скрытого элемента. Ведь если нормальное отображение так или иначе нарушится при выполнении функции выборки размеров скрытого элемента, то рассматриваемый здесь прием никуда не годится.

В функции обратного вызова таймера сначала получается ссылка на субъекты тестирования (оба изображения), а затем утверждается, что размеры видимого изображения могут быть получены из свойств `offsetHeight` и `offsetWidth`. Соответствующие тесты проходят, как показано на рис. 12.5.

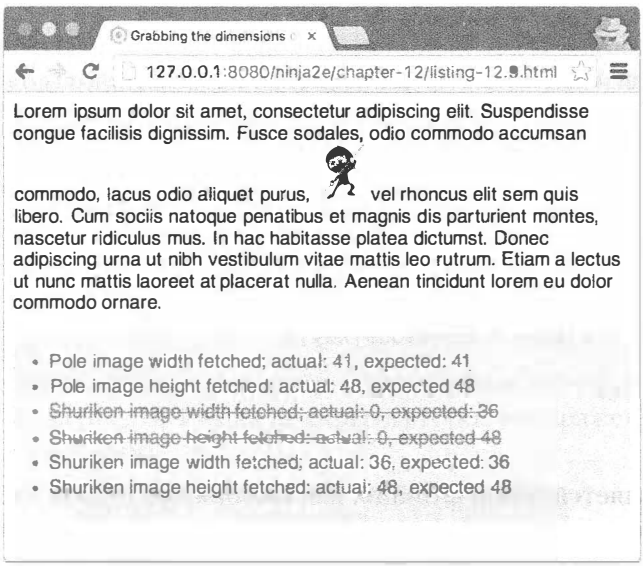


Рис. 12.5. Временно корректируя свойства стилевого оформления скрытых элементов разметки, можно успешно выбирать их размеры

Далее тот же самый тест выполняется для скрытого элемента разметки. В этом тесте неверно утверждается, что в свойствах `offsetHeight` и `offsetWidth` будут доступны прежние значения. И нет ничего удивительного в том, что этот тест не проходит. Ведь мы уже убедились в этом раньше. После этого новая функция вызывается для скрытого изображения, а результаты этой операции проверяются еще раз. На этот раз тест проходит, как показано на рис. 12.5.

Наблюдая за отображением содержимого веб-страницы по ходу выполнения теста (напомним, что выполнение теста задерживается на 3 секунды после загрузки страницы и создания модели DOM), можно заметить, что отображение никоим образом не нарушают подспудные манипуляции свойствами стилевого оформления скрытого элемента разметки.

Совет

Проверка свойств `offsetWidth` и `offsetHeight` стилевого оформления на нулевые значения может послужить невероятно эффективным способом для определения видимости элемента разметки.

12.4. Сведение к минимуму перегрузки верстки

До сих пор в этой главе пояснялось, насколько относительно просто видоизменить модель DOM, создавая и встраивая новые элементы, удаляя существующие элементы или изменяя их атрибуты. Видоизменение модели DOM относится к числу самых основных средств для достижения высокой динамичности веб-приложений.

Но это средство применяется с оговорками, к числу которых относится принятие во внимание *перегрузки верстки*, которая происходит в том случае, если выполняется последовательный ряд операций чтения и записи в модель DOM. И в ходе этого процесса браузеру не разрешается оптимизировать верстку.

Прежде чем перейти к более подробному обсуждению, следует принять во внимание, что изменение одного элемента разметки (или его содержимого) совсем не обязательно окажет влияние только на него, а, скорее всего, вызовет целую лавину изменений. Так, если установить ширину одного элемента, то соответствующие изменения произойдут в порожденных, родственных и родительских элементах. Следовательно, всякий раз, когда вносятся изменения, браузеру приходится оценивать их последствия. И в некоторых случаях подобные изменения просто неизбежны. В то же время не стоит возлагать излишнее бремя на бедных браузеров, поскольку это сразу же сказывается на производительности веб-приложений.

Перерасчет верстки обходится недешево, и поэтому браузеры не спешат его делать, откладывая этот вопрос на самый последний момент. Они пытаются поставить в очередь как можно больше операций записи в модели DOM, чтобы выполнить их одним пакетом. А когда появляется операция, требующая обновления верстки, браузер неохотно подчиняется, выполняя все пакетные операции и, наконец, обновляя верстку.

Но иногда способ написания прикладного кода не дает браузеру достаточной свободы для выполнения различных видов оптимизации, вынуждая его произвести немало (возможно, излишних) перерасчетов. В этом, собственно, и состоит перегрузка верстки. Она возникает всякий раз, когда в прикладном коде последовательно выполняется целый ряд (зачастую ненужных) операций

чтения и записи в модели DOM, не дающих браузеру возможность оптимизировать операции верстки. Но дело в том, что всякий раз, когда видоизменяется модель DOM, браузеру приходится производить перерасчет верстки перед чтением любой информации об элементе. А такое действие обходится недешево с точки зрения производительности. Рассмотрим в качестве примера код из листинга 12.10.

Чтение значения свойства `clientWidth` каждого элемента относится к числу тех действий, которые требуют от браузера обновить верстку. Если пытаться выполнить последовательный ряд операций записи и чтения в свойстве `width` разных элементов, браузер не сможет благоразумно отложить их до удобного момента. Вместо этого ему придется всякий раз производить перерасчет верстки, чтобы убедиться в правильности информации об элементах разметки, поскольку она читается после каждого видоизменения верстки.

Листинг 12.10. Последовательный ряд операций чтения и записи приводит к перегрузке верстки

```
<div id="ninja">I'm a ninja</div>
<div id="samurai">I'm a samurai</div>
<div id="ronin">I'm a ronin</div>
<script>
  const ninja = document.getElementById("ninja");
  const samurai = document.getElementById("samurai");
  const ronin = document.getElementById("ronin");

  const ninjaWidth = ninja.clientWidth;
  ninja.style.width = ninjaWidth/2 + "px";

  const samuraiWidth = samurai.clientWidth;
  samurai.style.width = samuraiWidth/2 + "px";

  const roninWidth = ronin.clientWidth;
  ronin.style.width = roninWidth/2 + "px";
</script>
```

Создать ряд новых элементов HTML-разметки

Извлечь элементы из модели DOM

Выполнить последовательный ряд операций чтения и записи. Изменения в модели DOM делают верстку недействительной

Чтобы свести к минимуму перегрузку верстки, можно, в частности, написать код таким образом, чтобы исключить ее излишние перерасчеты. Например, исходный код из листинга 12.10 можно переписать так, как показано в листинге 12.11.

Листинг 12.11. Пакетный режим выполнения операций чтения и записи в модели DOM, позволяющий избежать перегрузки верстки

```
<div id="ninja">I'm a ninja</div>
<div id="samurai">I'm a samurai</div>
<div id="ronin">I'm a ronin</div>
<script>
  const ninja = document.getElementById("ninja");
  const samurai = document.getElementById("samurai");
  const ronin = document.getElementById("ronin");
```

```
  const ninjaWidth = ninja.clientWidth;
  const samuraiWidth = samurai.clientWidth;
  const roninWidth = ronin.clientWidth;

  ninja.style.width = ninjaWidth/2 + "px";
  samurai.style.width = samuraiWidth/2 + "px";
  ronin.style.width = roninWidth/2 + "px";
</script>
```

Все операции чтения свойств элементов верстки собираются в пакет

Все операции записи свойств элементов верстки собираются в пакет

В данном примере кода все операции чтения и записи выполняются одним пакетом, поскольку между размерами элементов, как известно, отсутствуют зависимости. В частности, установка ширины элемента `ninja` не оказывает никакого влияния на ширину элемента `samurai`. Это дает браузеру возможность откладывать выполнение пакетных операций, видоизменяющих модель DOM.

Перегрузка верстки практически не заметна для более простых и мелких страниц, но ее все же необходимо иметь в виду при разработке сложных веб-приложений, особенно для мобильных устройств. Именно поэтому рекомендуется всегда иметь в виду методы и свойства, требующие обновления верстки, как показано в табл. 12.2, взятой из источника, доступного по адресу `http://ricostacruz.com/cheatsheets/layout-thrashing.html`.

Таблица 12.2. Интерфейсы API и свойства, делающие верстку недействительной

Интерфейс	Свойства
`Element`	`clientHeight`, `clientLeft`, `clientTop`, `clientWidth`, `focus`, `getBoundingClientRect`, `getClientRects`, `innerText`, `offsetHeight`, `offsetLeft`, `offsetParent`, `offsetTop`, `offsetWidth`, `outerText`, `scrollByLines`, `scrollByPages`, `scrollHeight`, `scrollIntoView`, `scrollIntoViewIfNeeded`, `scrollLeft`, `scrollTop`, `scrollWidth`
`MouseEvent`	`layerX`, `layerY`, `offsetX`, `offsetY`
`Window`	`getComputedStyle`, `scrollBy`, `scrollTo`, `scroll`, `scrollY`
`Frame`, `Document`, `Image`	`height`, `width`

Разработан ряд библиотек, в которых предпринята попытка свести к минимуму перегрузку верстки. К числу самых распространенных относится библиотека FastDom (`https://github.com/wilsonpage/fastdom`). В информационном хранилище этой библиотеки содержатся примеры, наглядно показывающие выигрыш в производительности, которого можно достичь в пакетном режиме выполнения операций чтения и записи в модели DOM (`https://wilsonpage.github.io/fastdom/examples/aspect-ratio.html`).

Виртуальная модель DOM библиотеки React

К числу самых распространенных относится клиентская библиотека React (**`https://facebook.github.io/react/`**) от Facebook. Высокая производительность библиотеки React была достигнута с помощью виртуальной модели DOM, состоящей из ряда объектов JavaScript, имитирующих фактическую модель

DOM. При разработке веб-приложений в React все изменения происходят в виртуальной модели DOM безотносительно к перегрузке верстки. А в подходящий момент виртуальная модель DOM применяется в библиотеке React с целью выяснить, какие изменения должны быть сделаны в фактической модели DOM, чтобы сохранить синхронизованным пользовательский интерфейс. Такой способ выполнения обновлений в пакетном режиме заметно повышает производительность веб-приложений.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Чтобы преобразовать HTML-строки в элементы модели DOM, необходимо выполнить следующие действия.
  - Убедиться, что HTML-строка содержит корректный код HTML-разметки.
  - Заключить HTML-строку в контейнер, требующийся согласно правил, принятым в браузере.
  - Вставить HTML-разметку в фиктивный элемент модели DOM через свойство `innerHTML` элемента модели DOM.
  - Извлечь созданные узлы модели DOM в обратном порядке.
- Для быстрой вставки узлов служат фрагменты модели DOM, поскольку фрагмент может быть вставлен в течение одной операции. И благодаря этому значительно сокращается количество операций.
- Атрибуты и свойства элемента модели DOM не всегда одно и то же, хотя и взаимосвязаны! Читать и записывать значения в атрибутах модели DOM можно с помощью методов `getAttribute()` и `setAttribute()`, тогда как запись в свойства модели DOM осуществляется с использованием обозначения свойства объекта (через точку).
- Обращаясь с атрибутами и свойствами, приходится иметь в виду *специальные атрибуты*. Те атрибуты, которые решено разместить в элементах HTML-разметки, чтобы передать полезную для приложений информацию, не представлены автоматически в качестве свойств элементов.
- Свойство `style` элемента является объектом со свойствами, соответствующими значениям стилевого оформления в разметке элемента. Для получения вычисленных стилей, в которых во внимание принимаются стили, заданные в таблицах стилей, служит метод `getComputedStyle()`.
- Для получения размеров отдельных элементов HTML-разметки служат свойства `offsetWidth` и `offsetHeight`.
- Перегрузка верстки происходит при выполнении в прикладном коде последовательного ряда операций чтения и записи в модели DOM. И вся-

кий раз браузер вынужден производить перерасчет информации о верстке. А это приводит к замедлению реакции веб-приложений.

- Собирайте свои обновления модели DOM в один пакет!

Упражнения

1. Какие утверждения пройдут в приведенном ниже фрагменте кода?

```
<div id="samurai"></div>
<script>
  const element = document.querySelector("#samurai");

  assert(element.id === "samurai", "property id is samurai");
  assert(element.getAttribute("id") === "samurai",
         "attribute id is samurai");

  element.id = "newSamurai";

  assert(element.id === "newSamurai", "property id is newSamurai");
  assert(element.getAttribute("id") === "newSamurai",
         "attribute id is newSamurai");
</script>
```

2. Каким из перечисленных ниже способов можно получить доступ к свойству `border-width` стилевого оформления элемента разметки в приведенном ниже фрагменте кода?

```
<div id="element" style="border-width: 1px;
                         border-style:solid; border-color: red">
</div>
<script>
  const element = document.querySelector("#element");
</script>
```

   а) `element.border-width`

   б) `element.getAttribute("border-width");`

   в) `element.style["border-width"];`

   г) `element.style.borderWidth;`

3. Какой из перечисленных ниже встроенных методов позволяет получить все стили, применяемые к определенному элементу (т.е. стили, предоставляемые браузером; стили, применяемые через таблицы стилей; а также свойства, устанавливаемые через атрибут стилевого оформления)?

   а) `getStyle()`

   б) `getAllStyles()`

   в) `getComputedStyle()`

4. Когда происходит перегрузка верстки?

13 Особенности обработки событий

В этой главе...

- Представление о цикле ожидания событий
- Решение сложных задач с помощью таймеров
- Управление анимацией с помощью таймеров
- Всплывание и делегирование событий
- Применение специальных событий

В главе 2 вкратце обсуждалась однопоточная модель выполнения кода в JavaScript, а также внедрение цикла ожидания событий и очереди событий, где события ожидают своего череда на обработку. Это обсуждение оказалось особенно полезным для описания отдельных шагов жизненного цикла веб-страницы, особенно порядка выполнения некоторых фрагментов кода JavaScript. В то же время это обсуждение оказалось упрощенным, и поэтому для более полного представления о механизме работы браузера бо́льшая часть этой главы посвящена исследованию особенностей цикла ожидания событий. Это поможет лучше понять некоторые ограничения, накладываемые на производительность приложений JavaScript, выполняемых в среде браузера. На основании знаний, почерпнутых из этой главы, вы сможете разрабатывать более плавно работающие веб-приложения.

В ходе этого исследования мы уделим особое внимание таймерам — языковому средству JavaScript, с помощью которого можно асинхронно задерживать на некоторое время выполнение фрагмента кода. На первый взгляд, проку от таймеров может показаться не особенно много, но в этой главе будет показано,

как пользоваться таймерами для разбиения длительных задач, замедляющих реакцию веб-приложений, на более мелкие задачи, не стопорящие работу браузеров. Умение пользоваться таймерами помогает в разработке высоко интерактивных веб-приложений.

Продолжая наше исследование, покажем, каким образом события распространяются по дереву модели DOM и как знание этого процесса помогает в написании более простого кода, меньше потребляющего ресурсы оперативной памяти. И в завершение главы мы рассмотрим особенности создания специальных событий, с помощью которых можно устранить тесную связь между различными частями приложения. Итак, без лишних церемоний приступим непосредственно к рассмотрению цикла ожидания событий.

Знаете ли вы?

Почему не гарантируется правильный хронометраж функций обратного вызова таймера?

Сколько раз функция обратного вызова таймера ставится в очередь микрозадач, если таймер, устанавливаемый с помощью функции `setInterval()`, срабатывает каждые 3 мс, тогда как параллельно выполняется другой обработчик события в течение 16 мс?

Почему контекст функции обработчика события иногда отличается от целевого объекта события?

13.1. Углубленное исследование цикла ожидания событий

Как упоминалось ранее, цикл ожидания событий намного сложнее, чем было показано в главе 2. Прежде всего, вместо единственной очереди, содержащей только события, у цикла ожидания событий имеется по меньшей мере две очереди, которые содержат, помимо событий, другие действия, выполняемые браузером. Такие действия называются *задачами* и разделяются на две категории: *макрозадачи* (зачастую называемые просто *задачами*) и *микрозадачи*.

К примерам макрозадач относится создание главного объекта документа, синтаксический анализ HTML-кода, выполнение основного (или глобального) кода JavaScript, изменение текущего URL, а также обработка различных событий, происходящих в результате загрузки страницы, ввода данных, ошибок в сети и срабатывания таймера. С точки зрения браузера макрозадача представляет собой отдельную, самостоятельную единицу работы. Выполнив одну задачу, браузер может продолжить выполнение других назначенных задач, например, повторный пересчет элементов пользовательского интерфейса на странице или сборку "мусора".

С другой стороны, микрозадачи представляют собой более мелкие задачи по обновлению состояния приложения, которые должны быть выполнены, прежде чем браузер продолжит выполнение других назначенных задач, напри-

мер, повторный пересчет элементов пользовательского интерфейса. К примерам микрозадач относятся функции обратного вызова обещаний и изменения в модели DOM. Микрозадачи должны выполняться как можно раньше и асинхронно, но и без больших затрат ресурсов, как на выполнение целой новой макрозадачи. Микрозадачи позволяют выполнять определенные действия *перед* повторным пересчетом элементов пользовательского интерфейса, чтобы избежать ненужных действий, которые могут выявить противоречивые состояния нашего приложения.

Примечание

В спецификации стандарта ECMAScript циклы ожидания событий вообще не упоминаются. Цикл ожидания событий подробно описывается в спецификации HTML (`https://html.spec.whatwg.org/#event-loops`), где обсуждаются также понятия макро- и микрозадач. В спецификации стандарта ECMAScript упоминаются *задания*, аналогичные микрозадачам, в связи с функциями обратного вызова обработки обещаний (`http://ecma-international.org/ecma-262/6.0/#sec-jobs-and-job-queues`). Несмотря на то что цикл ожидания событий определяется в спецификации HTML, он употребляется и в других средах, например, на платформе Node.js.

В реализации цикла ожидания событий должна использоваться по меньшей мере одна очередь для макрозадач и хотя бы еще одна для микрозадач. Но реализации цикла ожидания событий этим не ограничиваются и, как правило, предполагают наличие *нескольких* очередей макро- и микрозадач. Это дает возможность назначать приоритеты по типам задач в цикле ожидания событий, отдавая наибольший приоритет таким чувствительным ко времени реакции задачам, как ввод пользовательских данных. Но в связи с наличием многих браузеров и исполняющих сред JavaScript не следует удивляться, если встретятся циклы ожидания событий с единственной очередью для обоих типов задач.

Цикл ожидания событий основывается на следующих основополагающих принципах.

- Задачи обрабатываются по очереди.
- Задача выполняется до конца и не может быть прервана другой задачей.

Обратимся к рис. 13.1, где наглядно показаны оба принципа.

На рис. 13.1 схематически показан один шаг цикла ожидания событий, где сначала проверяется очередь макрозадач, и если имеется макрозадача, ожидающая своей очереди, то она выполняется. Как только задача будет полностью выполнена (или очередь задач опустеет), цикл ожидания событий перейдет к обработке очереди микрозадач. Если же в очереди имеется ожидающая микрозадача, она выполняется в цикле ожидания событий до полного завершения. В результате будут выполнены все микрозадачи в очереди. Обратите внимание на следующее отличие в обработке макро- и микрозадач: на одном шаге цикла, с одной стороны, выполняется не больше одной макрозадачи, тогда как остальные ожидают своей очереди, а с другой стороны — все микрозадачи.

А теперь, когда цикл ожидания событий разъяснен в общих чертах, выясним следующие интересные подробности (рис. 13.1).

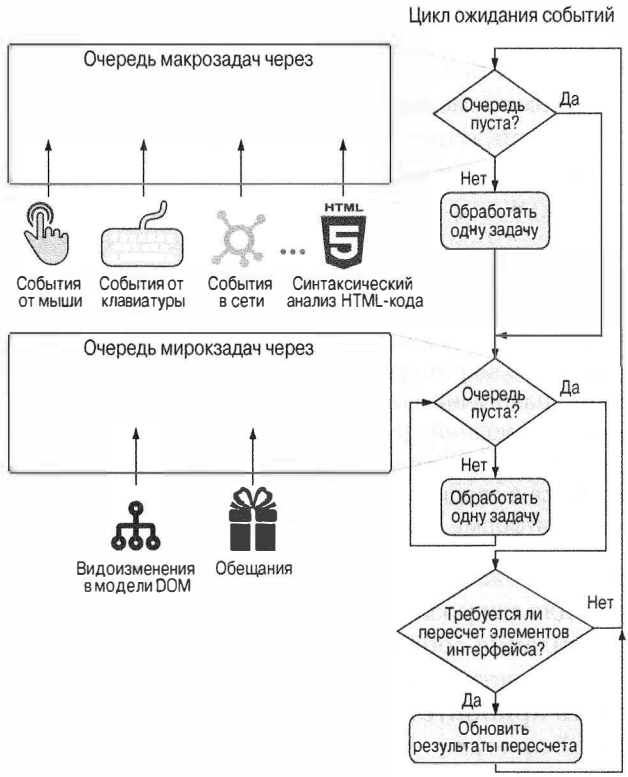


Рис. 13.1. В цикле ожидания событий доступны по меньшей мере две очереди: микро- и макрозадач. Оба типа задач обрабатываются поочередно

- **Обе очереди задач размещены за пределами цикла ожидания событий**, явно указывая на то, что действие по вводу задач в соответствующие очереди происходит за пределами цикла ожидания событий. Ведь иначе будут проигнорированы любые события, происходящие во время выполнения кода задачи JavaScript. А поскольку этого никак нельзя допустить, то действия по обнаружению и вводу задач выполняются отдельно от цикла ожидания событий.
- **Оба типа задач выполняются по очереди**, поскольку в JavaScript используется однопоточная модель выполнения кода. Если выполнение некоторой задачи началось, оно не может быть прервано другой задачей и должно быть доведено до ее полного завершения. Однако браузер может принудительно остановить выполнение задачи, например, в том случае, если начатая задача отнимает слишком много времени или памяти.

- **Все микрозадачи должны выполняться перед очередным пересчетом элементов интерфейса**, поскольку их цель — обновить состояние приложения перед пересчетом.
- **Браузер обычно пытается выполнить пересчет элементов интерфейса веб-страницы 60 раз в секунду**, чтобы получить частоту обновления 60 кадров в секунду, которая обычно считается идеальной для плавного воспроизведения анимации. Это означает, что браузер пытается воспроизвести 1 кадр через каждые 16,7 мс. Следует заметить, что действие "Обновить результаты пересчета", приведенное на рис. 13.1, происходит в цикле ожидания событий, поскольку содержимое страницы не должно видоизменяться другой задачей во время пересчета элементов интерфейса веб-страницы. Следовательно, если требуется добиться плавного выполнения приложений, то на обработку задач на одном шаге цикла ожидания событий остается не так уж и много времени. Одна задача и все составляющие ее микрозадачи в идеальном случае *должны* быть завершены в течение 16,7 мс.

А теперь рассмотрим три ситуации, которые могут возникнуть на следующем шаге цикла ожидания событий после того, как браузер завершит пересчет элементов интерфейса веб-страницы.

- Цикл ожидания событий достигает момента принятия решения "Требуется ли пересчет элементов интерфейса?", прежде чем завершится очередной промежуток времени в 16,7 мс. Обновление пользовательского интерфейса является сложной операцией, и поэтому браузер может отказаться от его выполнения, если не было запроса на явный пересчет элементов интерфейса.
- Цикл ожидания событий достигает момента принятия решения "Требуется ли пересчет элементов интерфейса?" приблизительно через 16 мс после предыдущего пересчета. В этом случае браузер обновляет пользовательский интерфейс, а пользователи увидят плавный ход выполнения приложения.
- Для выполнения следующей задачи (и всех связанных с ней микрозадач) потребуется намного больше, чем 16 мс. В этом случае браузер не сможет пересчитать элементы интерфейса страницы с требуемой частотой, и пользовательский интерфейс останется не обновленным. Если, с одной стороны, для выполнения кода задачи потребуется не слишком много времени (т.е. не больше 100–200 миллисекунд), такая задержка может оказаться совсем незаметной, особенно если на странице не происходит интенсивного обновления. А если, с другой стороны, на пересчет элементов интерфейса страницы потребуется слишком много времени, либо на странице будет выполняться сложная анимация, то пользователи, скорее всего, воспримут такую страницу как слишком медленно реагирующую на их действия. В худшем случае, когда задача выполняется

больше двух секунд, браузер пользователя выдаст неприятное сообщение "Unresponsive script" (Сценарий перестал отвечать на запросы). (Далее в этой главе будет показано, как разбивать сложные задачи на более мелкие, не стопорящие цикл ожидания событий.)

Примечание

Будьте внимательны при выборе событий для обработки, учитывая частоту их наступления и время, требующееся для их обработки. Особое внимание следует уделить организации обработки событий от мыши. Перемещение мыши приводит к размещению в очереди целого ряда событий, и поэтому выполнение любой сложной операции в обработчике событий от мыши — это прямой путь к написанию медленного и работающего рывками веб-приложения.

Итак, описав принцип, по которому действует цикл ожидания событий, перейдем к подробному рассмотрению ряда примеров его реализации.

13.1.1. Пример только с очередью макрозадач

Однопоточная модель выполнения кода в JavaScript неизбежно приводит к тому, что одновременно может быть выполнена только одна задача. А это, в свою очередь, означает, что все созданные задачи вынуждены ожидать до тех пор, пока не настанет их очередь на выполнение.

Рассмотрим пример простой веб-страницы, содержащей следующее.

- Нетривиальный основной (т.е. глобальный) код JavaScript.
- Две экранные кнопки и два нетривиальных обработчика событий от щелчков кнопками мыши — по одному на каждую кнопку.

Соответствующий пример кода приведен в листинге 13.1.

Листинг 13.1. Псевдокод для демонстрации цикла ожидания событий с одной очередью задач

```
<button id="firstButton"></button>
<button id="secondButton"></button>
<script>
  const firstButton = document.getElementById("firstButton");
  const secondButton = document.getElementById("secondButton");
  firstButton.addEventListener("click", function firstHandler(){
    /* Код обработчика события от щелчка кнопкой мыши,
       выполняющийся в течение 8 мс */
  });
  secondButton.addEventListener("click", function secondHandler(){
    /* Код обработчика события от щелчка кнопкой мыши,
       выполняющийся в течение 5 мс */
  });
  /* Код, выполняющийся в течение 15 мс */
</script>
```

Зарегистрировать обработчик событий от щелчков на первой экранной кнопке

Зарегистрировать еще один обработчик событий, на этот раз от щелчков на второй экранной кнопке

Для рассмотрения данного примера кода потребуется немного воображения. И вместо того чтобы загромождать данный пример излишним кодом, допустим следующее.

- Для выполнения основного кода JavaScript потребуется 15 мс.
- Для выполнения обработчика события от щелчка на первой экранной кнопке потребуется 8 мс.
- Для выполнения обработчика события от щелчка на второй экранной кнопке потребуется 5 мс.

А теперь представьте себе сверхшустрого пользователя, умудряющегося щелкнуть на первой экранной кнопке через 5 мс после начала выполнения сценария из рассматриваемого здесь примера, а на второй экранной кнопке — через 12 мс. Подобная ситуация наглядно показана на рис. 13.2.

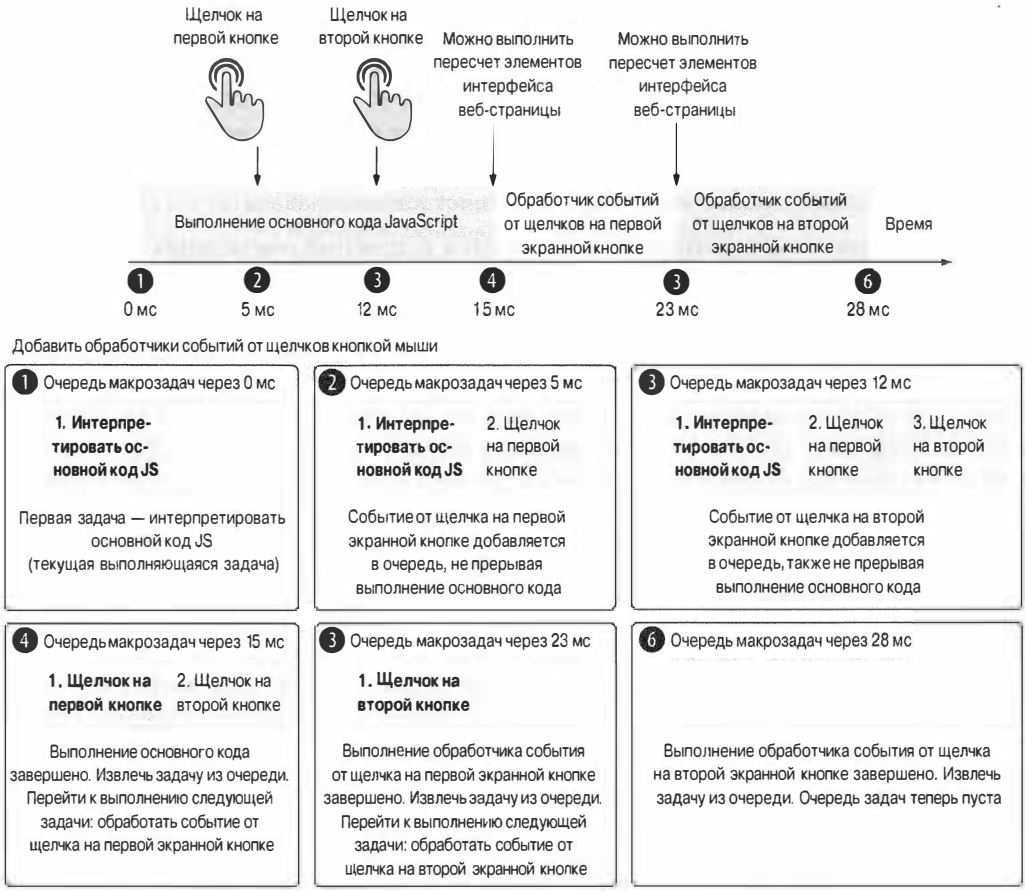


Рис. 13.2. На этой временной диаграмме наглядно показано, каким образом события помещаются в очередь задач по мере их наступления. Как только выполнение задачи завершится, она будет удалена из очереди, а цикл ожидания событий перейдет к выполнению следующей задачи

Из рис. 13.2 можно извлечь немало полезной информации, и если уяснить ее полностью, то можно лучше понять, каким образом действует цикл ожидания событий. В верхней части рис. 13.2 показана временная шкала (в миллисекундах), проведенная по оси X слева направо. А в расположенных ниже прямоугольных рамках представлены отдельно выполняемые блоки кода JavaScript с пояснением действий, которые происходят в соответствующие моменты времени. Например, первый блок основного кода JavaScript выполняется в течение 15 мс, обработчик события от щелчка на первой экранной кнопке — в течение 8 мс, а обработчик события от щелчка на второй экранной кнопке — в течение 5 мс. На временной диаграмме наглядно показано, когда наступают определенные события. Например, щелчок на первой экранной кнопке происходит через 5 мс после запуска приложения на выполнение, а щелчок на второй экранной кнопке — через 12 мс. В нижней части рис. 13.2 показано состояние очереди макрозадач в различные моменты выполнения прикладного кода.

Вначале рассматриваемого здесь примера выполняется основной код JavaScript. Из модели DOM немедленно выбираются два элемента, `firstButton` и `secondButton`, а две функции, `firstHandler()` и `secondHandler()`, регистрируются в качестве обработчиков событий от щелчков кнопками мыши:

```
firstButton.addEventListener("click", function firstHandler(){...});
secondButton.addEventListener("click", function secondHandler(){...});
```

Далее следует код, который выполняется в течение очередных 15 мс. И в этот промежуток времени наш шустрый пользователь щелкает сначала на первой экранной кнопке через 5 мс после запуска прикладной программы, а затем на второй экранной кнопке — через 12 мс.

Как уже не раз отмечалось, в языке JavaScript используется однопоточная модель выполнения кода, и поэтому щелчок на первой экранной кнопке совсем не означает, что его обработчик будет выполнен немедленно. (Напомним, что если одна задача уже выполняется, ее не может прервать другая задача.) Вместо этого событие, связанное со щелчком на первой экранной кнопке, размещается в очереди задач, где оно терпеливо ожидает своей обработки. То же самое происходит, когда пользователь щелкает на второй экранной кнопке; соответствующее событие размещается в очереди задач, где оно ожидает своей обработки. Следует особенно подчеркнуть, что обнаружение и размещение события в очереди задач происходит за пределами цикла ожидания событий. Задачи помещаются в очередь задач даже во время выполнения основного кода JavaScript.

Если сделать моментальный снимок очереди задач через 12 мс после запуска на выполнение рассматриваемого здесь сценария, то обнаружатся следующие задачи.

- Выполнение основного кода JavaScript, т.е. текущей задачи.
- Щелчок на первой экранной кнопке, т.е. событие, наступающее после нажатия данной кнопки.

- Щелчок на второй экранной кнопке, т.е. событие, наступающее после нажатия данной кнопки.

Перечисленные выше задачи показаны также на рис. 13.3.

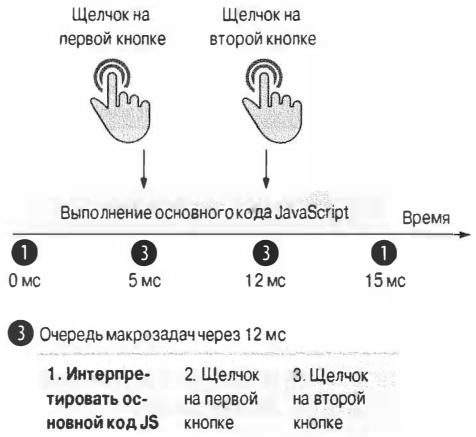


Рис. 13.3. Через 12 мс после запуска приложения на выполнение в очереди задач оказываются три задачи: одна — для выполнения основного кода JavaScript, т.е. текущая задача, и еще две задачи — по одной на каждое событие от щелчков кнопками мыши

Еще один интересный момент возникает через 15 мс после запуска на выполнение рассматриваемого здесь приложения, когда завершается выполнение основного кода JavaScript. Как показано на рис. 13.1, по завершении задачи цикл ожидания событий переходит к обработке очереди микрозадач. А поскольку в данный момент никаких микрозадач в очереди нет (она даже не показана на временной диаграмме, потому что пуста), то данный этап пропускается и происходит переход к обновлению пользовательского интерфейса. В данном примере обновление пользовательского интерфейса не обсуждается, хотя оно происходит и отнимает некоторое время. На этом первый шаг цикла ожидания событий завершается, и начинается второй шаг — переход к следующей по очереди задаче.

Далее начинается выполнение задачи обработки события от щелчка на первой экранной кнопке (`firstButton`). На рис. 13.4 показано состояние очереди задач через 15 мс после запуска рассматриваемого здесь приложения на выполнение. В частности, выполнение обработчика события от щелчка на первой экранной кнопке, т.е. функции `firstHandler()`, отнимает 8 мс и происходит без прерываний, тогда как событие от щелчка на второй экранной кнопке (`secondButton`) ожидает своей очереди на обработку.

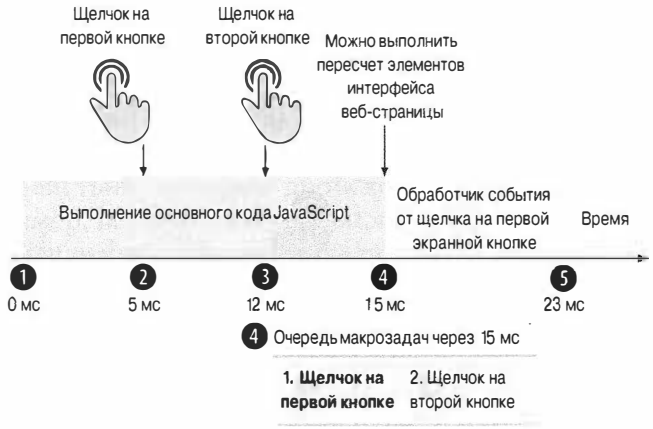


Рис. 13.4. Через 15 мс после запуска приложения на выполнение очередь задач содержит две задачи для обработки событий от щелчков кнопками мыши. И в этот момент выполняется первая задача

По истечении 23 мс событие от щелчка на первой экранной кнопке обрабатывается полностью, и соответствующая задача удаляется из очереди задач. Браузер снова опрашивает очередь микрозадач, которая по-прежнему пуста, и при необходимости повторяет пересчет элементов интерфейса веб-страницы.

И, наконец, на третьем шаге цикла ожидания событий обрабатывается событие от щелчка на второй экранной кнопке (`secondButton`), как показано на рис. 13.5. Выполнение обработчика события от щелчка на второй экранной кнопке, т.е. функции `secondHandler()`, отнимает около 5 мс, и, таким образом, очередь задач становится пустой через 28 мс после запуска данного приложения на выполнение.

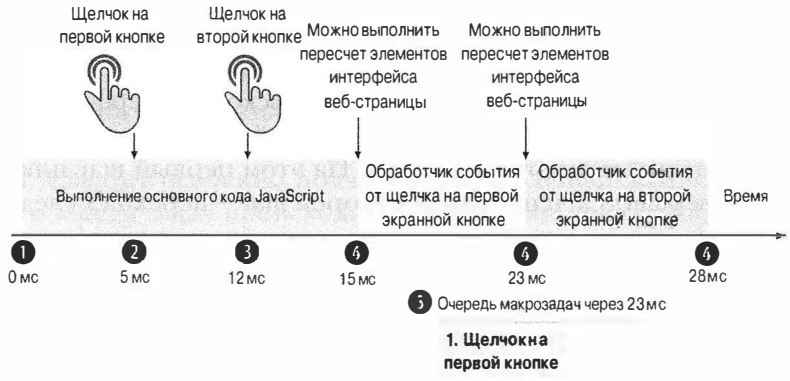


Рис. 13.5. Через 23 мс после запуска приложения на выполнение остается выполнить только одну задачу: обработать событие от щелчка на второй экранной кнопке

Данный пример наглядно показывает, что событие должно ожидать своей очереди на выполнение, если другие события уже обрабатываются. Например, обработчик события от щелчка на второй экранной кнопке вызывается лишь через 23 мс после запуска приложения на выполнение, хотя соответствующее событие наступило на 11 мс раньше. А теперь расширим данный пример, чтобы включить в него очередь микрозадач.

13.1.2. Пример с обеими очередями макро- и микрозадач

Мы выяснили, как работает цикл ожидания событий при наличии одной очереди задач. Теперь расширим рассматриваемый здесь пример, включив в него очередь микрозадач. Для этого проще всего включить обещание и код, обрабатывающий это обещание после его разрешения, в обработчик событий от щелчков на первой экранной кнопке. Как упоминалось в главе 6, *обещание* служит *заполнителем значения*, которого еще нет, но оно появится в дальнейшем. Обещание гарантирует, что в конечном итоге результат асинхронного вычисления станет известен. Именно поэтому обработчики обещаний, т.е. функции обратного вызова, указываемые в параметрах метода `then()` обещания, всегда вызываются асинхронно – даже если обещания уже будут разрешены.

В листинге 13.2 демонстрируется видоизмененный пример кода с двумя очередями задач.

Листинг 13.2. Псевдокод для демонстрации цикла ожидания событий с двумя очередями задач

```
<button id="firstButton"></button>
<button id="secondButton"></button>
<script>
  const firstButton = document.getElementById("firstButton");
  const secondButton = document.getElementById("secondButton");
  firstButton.addEventListener("click", function firstHandler(){
    Promise.resolve().then(() => {
      /* Код обработчика обещания, выполняющийся в течение 4 мс */
    });

    /* Код обработчика события от щелчка кнопкой мыши,
       выполняющийся в течение 8 мс */
  });
  secondButton.addEventListener("click", function secondHandler(){
    /* Код обработчика события от щелчка кнопкой мыши,
       выполняющийся в течение 5 мс */
  });
  /* Код, выполняющийся в течение 15 мс */
</script>
```

Немедленно разрешить обещание и передать функцию обратного вызова методу `then()`

В данном примере кода допускаются те же действия, что и в первом примере.

- Щелчок на первой экранной кнопке через 5 мс после запуска приложения на выполнение.
- Щелчок на второй экранной кнопке через 12 мс после запуска приложения на выполнение.
- Выполнение обработчика события от щелчка на первой экранной кнопке в течение 8 мс.
- Выполнение обработчика события от щелчка на второй экранной кнопке в течение 5 мс.

Единственное отличие на этот раз заключается в том, что в коде обработчика события от щелчка на первой экранной кнопке создается также немедленно разрешаемое обещание, которому передается функция обратного вызова, выполняемая в течение 4 мс. А поскольку обещание представляет будущее значение, которое обычно неизвестно сразу, то обещания всегда обрабатываются асинхронно.

Откровенно говоря, в данном случае, где создается немедленно разрешаемое обещание, интерпретатор JavaScript может сразу же сделать обратный вызов, поскольку уже известно, что обещание успешно разрешено. Но ради согласованности интерпретатор JavaScript вместо этого делает все обратные вызовы обещаний асинхронно после того, как будет выполнена остальная часть кода обработчика события от щелчка на первой экранной кнопке, что отнимает 8 мс. С этой целью создается новая микрозадача, которая затем размещается в очереди микрозадач. Исследуем временную диаграмму выполнения кода из данного примера, приведенную на рис. 13.6.

Временная диаграмма, приведенная на рис. 13.6, подобна временной диаграмме, приведенной на рис. 13.5. Если сделать моментальный снимок очереди задач через 12 мс после запуска приложения на выполнение, то в очереди обнаружатся практически те же самые задачи. Основной код JavaScript выполняется в то время, как задачи обработки событий от щелчков на первой и второй экранной кнопке ожидают своей очереди (как и на рис. 13.3). Но, помимо очереди задач, в данном примере основное внимание уделяется также очереди микрозадач, которая все еще остается пустой через 12 мс после запуска приложения на выполнение.

Следующий интересный момент в выполнении рассматриваемого здесь приложения наступает через 15 мс после его запуска, когда завершается выполнение основного кода JavaScript. В связи с завершением задачи в цикле ожидания событий проверяется очередь микрозадач, которая оказывается пустой, и тогда по мере надобности выполняется пересчет элементов интерфейса страницы. В данном случае фрагмент кода пересчета страницы ради простоты не включен во временную диаграмму на рис. 13.6.

На следующем шаге цикла ожидания событий выполняется задача обработки события от щелчка на первой экранной кнопке:

```
firstButton.addEventListener("click", function firstHandler(){
    Promise.resolve().then(() => {
```

```
    /* Код обработчика обещания, выполняющийся в течение 4 мс */
  });

  /* Код обработчика события от щелчка кнопкой мыши,
     выполняющийся в течение 8 мс */
});
```

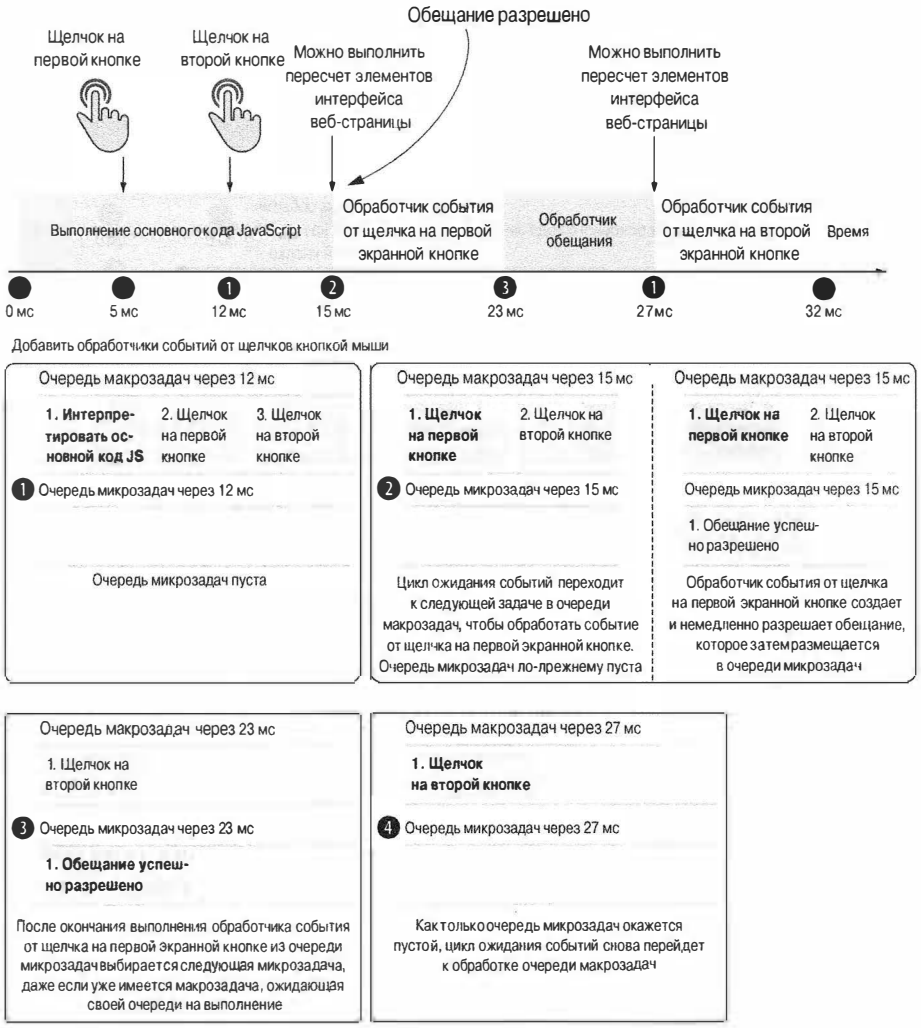


Рис. 13.6. Если микрозадача размещается в очереди микрозадач, она получает приоритет и запускается на выполнение даже в том случае, если ранее поставленная в очередь макрозадача уже ожидает своей очереди на выполнение. И в этом случае микрозадача успешного разрешения обещания получает приоритет над задачей обработки события от щелчка на второй экранной кнопке

В функции `firstHandler()` создается уже разрешенное обещание, для чего специально вызывается метод `Promise.resolve()` с функцией обратного вы-

зова, которая, без сомнения, будет вызвана, поскольку обещание уже разрешено. В итоге создается новая микрозадача для выполнения кода обратного вызова. Эта микрозадача размещается в очереди микрозадач, а обработчик событий от щелчка кнопкой мыши продолжает свое выполнение в течение еще 8 мс. Текущее состояние обеих очередей задач наглядно показано на рис. 13.7.

Рис. 13.7. Во время выполнения обработчика события от щелчка на первой экранной кнопке разрешается созданное обещание. В итоге микрозадача успешно разрешенного обещания размещается в очереди микрозадач и будет выполнена как можно скорее, но без прерывания выполнения текущей задачи

Вернемся к обеим очередям задачи через 23 мс после запуска на выполнение рассматриваемого здесь приложения. В этот момент событие от щелчка на первой экранной кнопке полностью обработано, а его задача удалена из очереди задач.

Для обработки в цикле ожидания событий должна быть выбрана следующая задача. В данный момент имеется одна макрозадача для обработки события от щелчка на второй экранной кнопке, размещенная в очереди задач через 12 мс после начала выполнения данного приложения, а также одна микрозадача для обработки успешно разрешенного обещания, размещенная в очереди микрозадач через 15 мс после запуска данного приложения на выполнение.

В данном случае по логике развития событий должна начать выполняться задача обработки события от щелчка на второй экранной кнопке, но, как упоминалось ранее, микрозадачи как более мелкие задачи должны выполняться как можно скорее. Приоритет отдается микрозадачам, и если вернуться к рис. 13.1, то можно заметить, что всякий раз, перед началом выполнения задачи, в цикле ожидания событий сначала проверяется очередь микрозадач с целью обработать все микрозадачи, прежде чем перейти к пересчету элементов интерфейса или другой задаче.

Именно по этой причине задача обработки успешно разрешенного обещания выполняется *сразу* после обработки события от щелчка на первой экранной кнопке, несмотря на то, что прежняя задача обработки события от щелчка на второй экранной кнопке все еще ожидает своей очереди на выполнение (рис. 13.8).

Рис. 13.8. Как только задача будет выполнена, в цикле ожидания событий обрабатываются все микрозадачи из очереди микрозадач. В данном случае выполняется микрозадача обработки успешно разрешенного обещания перед переходом к макрозадаче обработки события от щелчка на второй экранной кнопке

Следует особо подчеркнуть, что, как только макрозадача будет выполнена, цикл ожидания событий сразу же перейдет к обработке содержимого очереди микрозадач. При этом пересчет элементов интерфейса веб-страницы не выполняется до тех пор, пока очередь микрозадач не опустеет. В этом нетрудно убедиться, взглянув на временную диаграмму, приведенную на рис. 13.9.

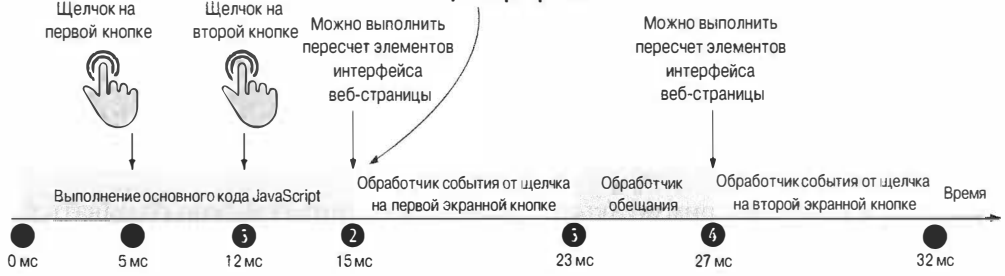


Рис. 13.9. Пересчет элементов интерфейса веб-страницы может быть выполнен в промежутке между макрозадачами (выполнения основного кода JavaScript и обработки события от щелчка на первой экранной кнопке), но перед этим должны быть выполнены все микрозадачи (т.е. до обработки обещания)

Как показано на рис. 13.9, пересчет элементов интерфейса веб-страницы может произойти в промежутке между двумя макрозадачами только в том случае, если в этом промежутке отсутствуют микрозадачи. В данном случае страница

может быть пересчитана в промежутке между выполнением основного кода JavaScript и обработкой события от щелчка на первой экранной кнопке. Но пересчет *нельзя* выполнить сразу после обработки события от щелчка на первой экранной кнопке, поскольку приоритет отдается микрозадачам и, в данном случае, обработке обещания.

Пересчет веб-страницы может произойти и после выполнения микрозадачи, но только в том случае, если отсутствуют другие микрозадачи, ожидающие своей очереди на выполнение. В данном примере браузер сможет пересчитать веб-страницу сразу после обработки обещания, но перед тем, как в цикле ожидания событий произойдет переход к обработке события от щелчка на второй экранной кнопке.

Следует, однако, иметь в виду, что микрозадаче обработки успешно разрешенного обещания ничто не мешает разместить в очереди другие микрозадачи, и всем этим микрозадачам будет отдан приоритет перед задачей обработки события от щелчка на второй экранной кнопке. При этом страница будет пересчитана снова, а переход к задаче обработки события от щелчка на второй экранной кнопке произойдет в цикле ожидания событий только после того, как опустеет очередь микрозадач, поэтому будьте внимательны!

А теперь, когда стало понятно, каким образом функционирует цикл ожидания событий, перейдем к исследованию особой группы событий, называемой таймерами.

13.2. Овладение таймерами: тайм-ауты и интервалы времени

Таймеры относятся к нередко злоупотребляемым и плохо понимаемым языковым средствам JavaScript. Но если правильно использовать таймеры, они способны улучшить разработку сложных приложений. Таймеры позволяют задерживать выполнение фрагмента кода, по крайней мере, на определенное количество миллисекунд. И этой особенностью таймеров можно воспользоваться, чтобы разбить длительные задачи на более мелкие и не загромождать цикл ожидания событий, который не позволяет браузеру выполнять пересчет элементов интерфейса веб-страницы, что в свою очередь замедляет реакцию веб-приложения.

Но прежде рассмотрим функции, с помощью которых можно создавать таймеры и манипулировать ими. Для создания таймеров в браузере предоставляются два метода, `setTimeout()` и `setInterval()`, а для сброса (или удаления) таймеров — еще два метода, `clearTimeout()` и `clearInterval()`. Все эти методы относятся к объекту `window`, представляющему глобальный контекст. Подобно циклу ожидания событий, таймеры определяются не в самом языке JavaScript, а реализуются в среде исполняющей среды (например, в браузере, действующем на стороне клиента, или на платформе Node.js, действующей на стороне сервера). В табл. 13.1 перечислены методы, предназначенные для создания и сброса таймеров.

Таблица 13.1. Методы манипулирования таймерами в JavaScript (все они относятся к глобальному объекту `window`)

Метод	Формат	Описание
`setTimeout()`	`id = setTimeout(fn,delay)`	Инициирует таймер, запускающий передаваемую ему функцию обратного вызова по истечении указанного времени задержки. Возвращает уникальное значение, позволяющее идентифицировать таймер
`clearTimeout()`	`clearTimeout(id)`	Удаляет (сбрасывает) таймер, идентифицируемый передаваемым значением, если таймер еще не сработал
`setInterval()`	`id = setInterval(fn,delay)`	Инициирует таймер на постоянный запуск передаваемой функции обратного вызова через указанный интервал времени вплоть до отмены. Возвращает уникальное значение, позволяющее идентифицировать таймер
`clearInterval()`	`clearInterval(id)`	Удаляет (сбрасывает) интервальный таймер, идентифицируемый передаваемым значением

Перечисленные выше методы позволяют устанавливать и сбрасывать таймеры, которые запускаются однократно или периодически через указанные промежутки времени. На практике в большинстве браузеров допускается сбрасывать оба вида таймеров методом `clearTimeout()` или `clearInterval()`. Но ради ясности эти методы рекомендуется применять согласованными парами.

Примечание

Очень важно понять, что задержка срабатывания таймера *не* гарантируется. Это в значительной степени связано с циклом ожидания событий, как поясняется в следующем разделе.

13.2.1. Запуск обработчиков таймера в цикле ожидания событий

Ранее мы подробно исследовали, что происходит при наступлении события. Но таймеры отличаются от стандартных событий, поэтому рассмотрим пример, аналогичный представленным ранее. Соответствующий код приведен в листинге 13.3.

Листинг 13.3. Псевдокод для демонстрации примера с тайм-аутом и интервальным таймером

```
<button id="myButton"></button>
<script>
  setTimeout(function timeoutHandler(){
    /* Код обработчика тайм-аута, выполняющийся
       в течение 6 мс */
  }, 10);
```

Зарегистрировать тайм-аут, истекающий через 10 мс

```
  setInterval(function intervalHandler(){
    /* Код обработчика интервального таймера, выполняющийся
       в течение 8 мс */
  }, 10);

  const myButton = document.getElementById("myButton");
  myButton.addEventListener("click", function clickHandler(){
    /* Код обработчика события от щелчка кнопкой мыши,
       выполняющийся в течение 10 мс */
  });
  /* Код, выполняющийся в течение 18 мс */
</script>
```

Зарегистрировать интервал времени, истекающий через каждые 10 мс

Зарегистрировать обработчик события от щелчка на экранной кнопке

В данном примере кода определяется только одна кнопка, но на этот раз регистрируются также два таймера. Прежде всего, регистрируется таймер тайм-аута, срабатывающий через 10 мс:

```
setTimeout(function timeoutHandler(){
  /* Код обработчика тайм-аута, выполняющийся
     в течение 6 мс */
}, 10);
```

Для обработки тайм-аута имеется функция обратного вызова, выполнение которой длится 6 мс. Далее регистрируется интервальный таймер, срабатывающий через каждые 10 мс:

```
setInterval(function intervalHandler(){
  /* Код обработчика интервального таймера, выполняющийся
     в течение 8 мс */
}, 10);
```

И в завершение данного примера выполняется блок кода приблизительно в течение 18 мс. (Вообразите, что на его месте выполняется сложный фрагмент кода.)

А теперь допустим, что и в данном случае нам приходится иметь дело с шустрым и нетерпеливым пользователем, который щелкает на экранной кнопке всего через 6 мс после запуска на выполнение рассматриваемого здесь приложения. Временная диаграмма выполнения данного приложения в течение 18 мс представлена на рис. 13.10.

Как и в предыдущих примерах, в первой задаче, находящейся в очереди, выполняется основной код JavaScript. В ходе этого выполнения, на которое уходит 18 мс, происходят следующие важные события.

1. В момент времени 0 мс запускается таймер тайм-аута на 10 мс, а также интервальный таймер, срабатывающий через каждые 10 мс. Ссылки на них сохраняются в браузере.
2. В момент времени 6 мс выполняется щелчок кнопкой мыши.
3. В момент времени 10 мс срабатывает таймер тайм-аута и первый раз — интервальный таймер.

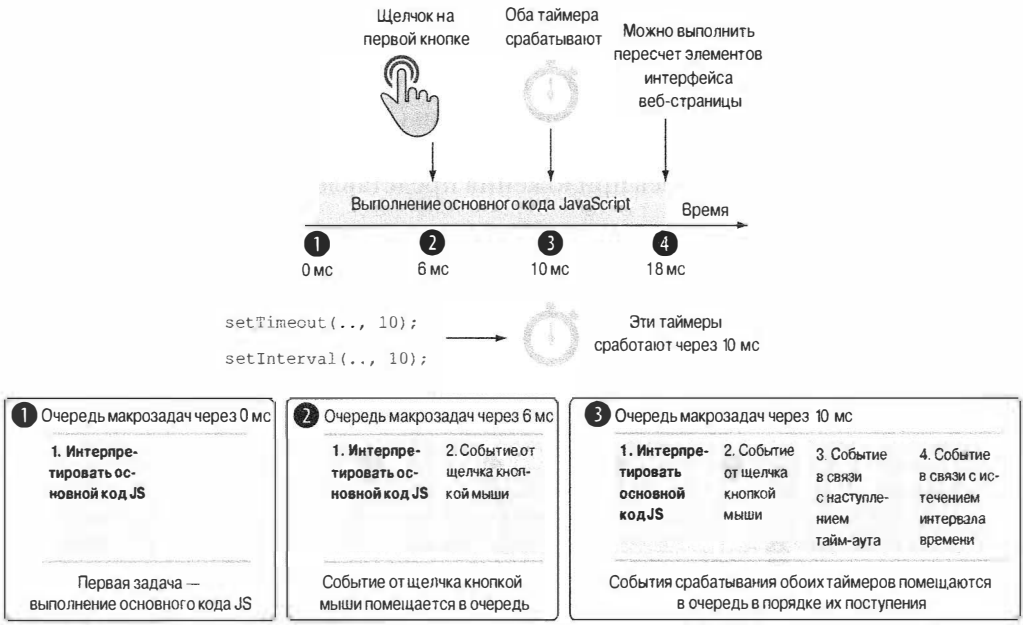


Рис. 13.10. Временная диаграмма выполнения прикладной программы в течение 18 мс. Первой задачей в данный момент является выполнение основного кода JavaScript, на что уходит 18 мс. В ходе этого выполнения происходят следующие события: щелчок кнопкой мыши, срабатывание таймера и истечение интервала времени

Как вам должно быть уже известно из предыдущего исследования цикла ожидания событий, каждая задача всегда выполняется до полного ее завершения и не может быть прервана другой задачей. В связи с этим все вновь созданные задачи размещаются в очереди, где они терпеливо ожидают своего череда на обработку. Когда пользователь щелкает на экранной кнопке через 6 мс после запуска рассматриваемого здесь приложения, соответствующая задача помещается в очередь задач. Следует иметь в виду, что оба таймера тайм-аута и интервала времени сработают через 10 мс после начала выполнения программы, и что после этого периода времени соответствующие им задачи будут помещены в очередь задач. Мы еще вернемся к данному вопросу, а пока достаточно сказать, что задачи помещаются в очередь в порядке регистрации их обработчиков событий: первым — обработчик событий при срабатывании таймера тайм-аута, вторым — обработчик событий при срабатывании интервального таймера.

Исходный блок кода завершает свое выполнение через 18 мс, а поскольку при этом отсутствуют микрозадачи, браузер может снова пересчитать элементы интерфейса веб-страницы и перейти ко второму шагу цикла ожидания событий, если ради простоты отложить на время вопросы хронометража. Состояние очереди задач в данный момент наглядно показано на рис. 13.11.

Спустя 18 мс, когда выполнение исходного блока кода будет завершено, в очередь на выполнение ставятся следующие три блока кода: обработчик собы-

тия от щелчка кнопкой мыши, обработчик события от срабатывании таймера тайм-аута и обработчик события от первого срабатывания интервального таймера. Это означает, что начинает выполняться ожидающий своей очереди обработчик события от щелчка кнопкой мыши (для ясности предположим, что это длится 10 мс). Очередная временная диаграмма выполнения в данный момент рассматриваемого здесь приложения представлена на рис. 13.12.

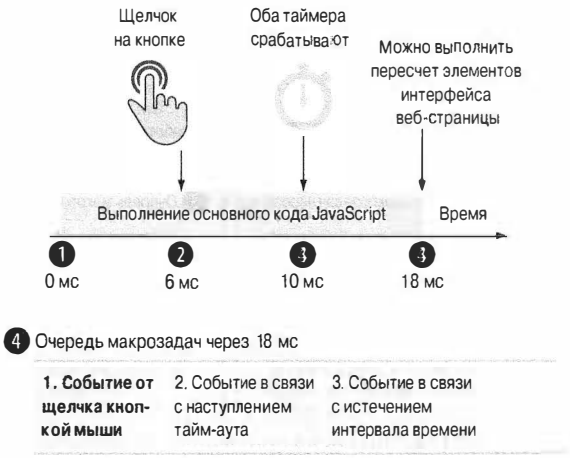


Рис. 13.11. При срабатывании таймеров соответствующие события размещаются в очереди задач

В отличие от функции `setTimeout()`, таймер которой срабатывает лишь один раз, таймер, созданный функцией `setInterval()`, будет срабатывать до тех пор, пока он не будет явно сброшен. Следовательно, приблизительно через 20 мс после запуска на выполнение рассматриваемого здесь приложения, интервальный таймер сработает в очередной раз. Как правило, это приводит к созданию новой задачи и ее помещению в очередь задач. Но теперь этого не произойдет, потому что экземпляр задачи обработки события в связи со срабатыванием интервального таймера уже находится в очереди, ожидая своего выполнения. Таким образом, *браузер не размещает в очереди больше одного экземпляра обработчика события в связи со срабатыванием интервального таймера.*

Выполнение обработчика события от щелчка кнопкой мыши завершается через 28 мс, после чего браузер снова может пересчитать веб-страницу, прежде чем цикл ожидания событий перейдет к следующему шагу, где выполняется задача обработки события в связи с наступлением тайм-аута. Но вернемся мысленно к началу данного примера, где для установки таймера тайм-аута, который должен сработать через 10 мс, вызывается следующая функция:

```
setTimeout(function timeoutHandler(){
  /* Код обработчика тайм-аута, выполняющийся
     в течение 6 мс */
}, 10);
```

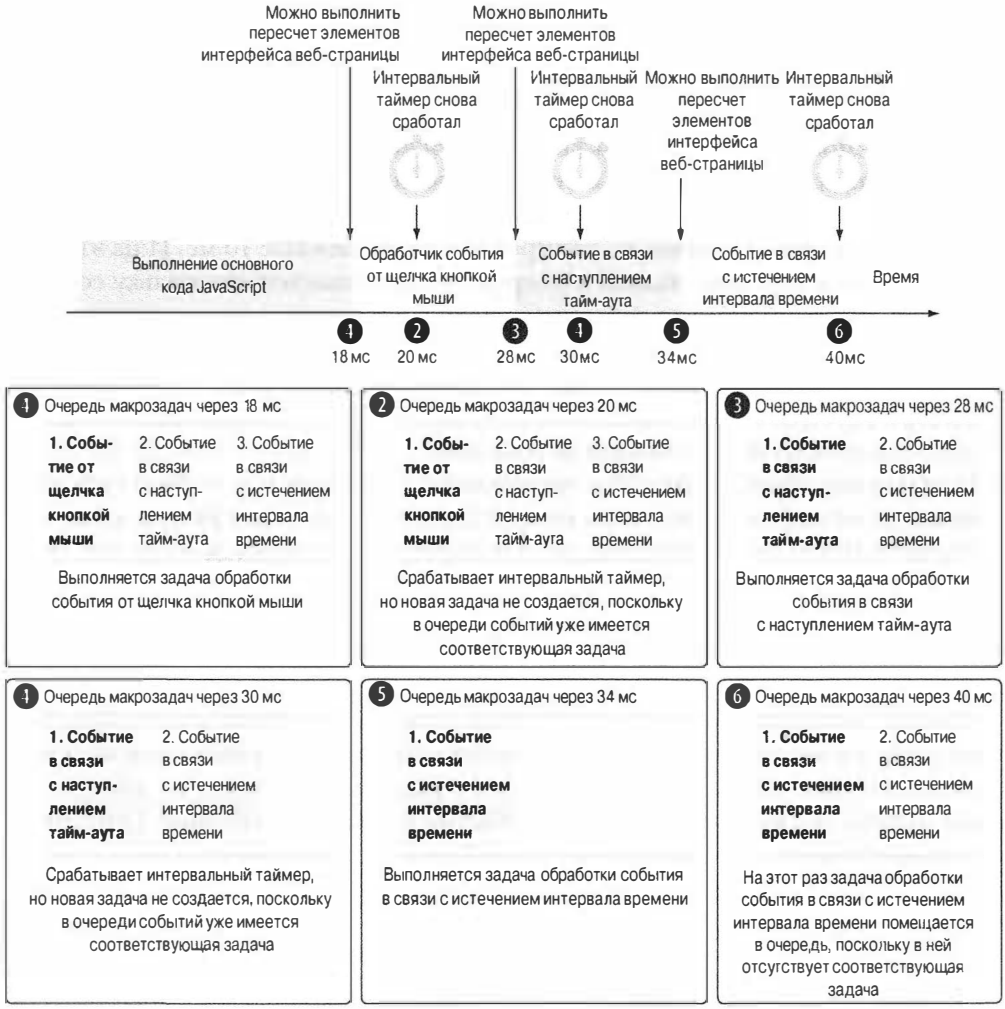


Рис. 13.12. Если событие наступает в связи с истечением интервала времени и связанная с ним задача уже находится в очереди ожидания, то новая задача не помещается в очередь и ничего не происходит, как следует из состояния очередей через 20 и 30 мс после запуска примера прикладной программы на выполнение

Поскольку это первая задача в нашем приложении, поэтому ничто не мешает предположить, что обработчик события в связи с истечением тайм-аута начнет выполняться ровно через 10 мс. Но, как следует из рис. 13.12, выполнение этого обработчика начинается аж через 28 мс!

Именно поэтому следует проявлять особую осторожность, утверждая, что таймер предоставляет возможность асинхронно задерживать выполнение фрагмента кода, по меньшей мере, на определенное количество миллисекунд. В силу однопоточного характера JavaScript контролю поддается только момент помещения в очередь задачи обработки события от срабатывания таймера, но

не момент ее фактического выполнения! Итак, прояснив эту небольшую проблему, продолжим исследование выполнения остальной части рассматриваемого приложения.

На выполнение задачи обработки события в связи с наступлением тайм-аута уходит 6 мс, поэтому она должна завершиться через 34 мс после запуска данного приложения на выполнение. В течение этого периода времени, когда наступает момент времени 30 мс, в очередной раз сработает интервальный таймер, поскольку его срабатывание запланировано через каждые 10 мс. И на этот раз никакая дополнительная задача в очередь не помещается, поскольку соответствующая задача обработки события в связи с истечением интервала времени уже ожидает своей очереди на выполнение. В момент времени 34 мс, когда завершается выполнение обработчика события в связи с истечением тайм-аута, у браузера снова появляется возможность пересчитать веб-страницу и перейти к следующему шагу цикла ожидания событий.

И, наконец, обработчик события в связи с истечением интервала времени начинает свое выполнение в момент времени 34 мс, т.е. через 24 мс *после* момента времени 10 мс, когда он был фактически помещен в очередь событий. И это лишний раз подчеркивает, что величина задержки, передаваемая в качестве аргумента при вызове функций `setTimeout(fn, delay)` и `setInterval(fn, delay)`, обозначает только задержку, после которой соответствующая задача помещается в очередь, а не конкретный момент ее выполнения.

Выполнение кода обработчика события в связи с истечением интервала времени длится 8 мс, поэтому в ходе его выполнения истекает еще один интервал времени в момент 40 мс. Но на этот раз в очередь задач наконец-то помещается новая задача, поскольку обработчик события в связи с истечением интервала времени выполняется, а не ожидает своей очереди. После этого выполнение данного приложения продолжается, как показано на рис. 13.13. Таким образом,

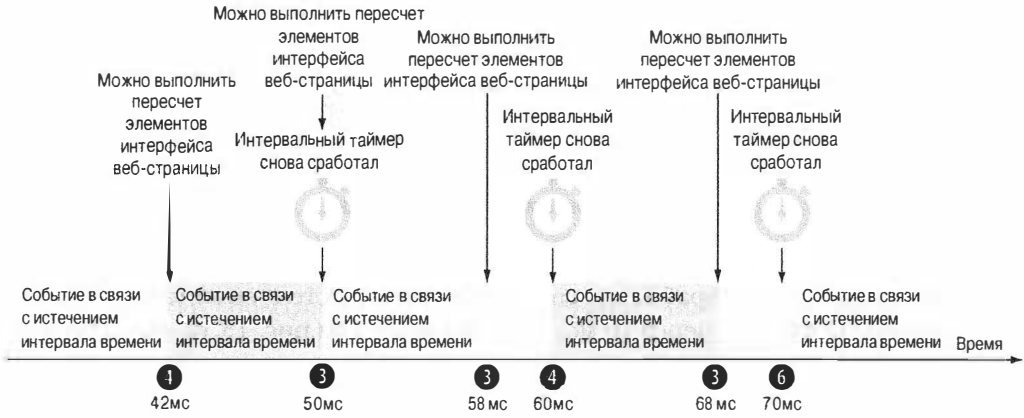


Рис. 13.13. Вследствие задержек, обусловленных обработкой событий от щелчков кнопкой мыши, и наступления тайм-аута, обработка событий от интервального таймера происходит не через каждые 10 мс, а с некоторым запаздыванием

установка задержки на 10 мс при вызове функции `setInterval()` совсем не означает, что данный обработчик события будет выполняться через каждые 10 мс. Например, обработчики событий от интервального таймера могут запускаться один за другим, как это и происходит в период между моментами времени 42 и 50 мс, поскольку соответствующие задачи уже были размещены в очереди, а продолжительность выполнения отдельных задач может варьироваться.

Наконец, через 50 мс после запуска данного приложения на выполнение ситуация нормализуется, и обработка событий в связи с истечением интервала времени происходит через каждые 10 мс. Из данного примера можно сделать следующий важный вывод: в цикле ожидания событий одновременно можно обработать только одно событие, выполнив до конца соответствующую задачу. И нет никакой гарантии, что обработчики событий при срабатывании таймеров будут выполняться именно тогда, когда это предполагается. И это особенно справедливо для обработчиков событий в связи с истечением интервала времени. Как было показано в данном примере, обратные вызовы выполнялись в моменты времени 34, 42, 50, 60 и 70 мс, хотя они и были запланированы в моменты времени 10, 20, 30, 40, 50, 60 и 70 мс. В данном случае по ходу выполнения нашей программы были полностью утрачены два события от интервального таймера, а обработчики некоторых событий были выполнены не тогда, когда это предполагалось.

Как видите, работа с интервальным таймером требует особого подхода, отличного от того, который применяется при работе с таймером тайм-аута. Рассмотрим этот подход более подробно.

Отличия тайм-аутов от интервалов времени

На первый взгляд, интервал времени может показаться похожим на тайм-аут, который периодически повторяется. Но их отличия более глубокие. Чтобы наглядно показать отличия в установке интервала времени и тайм-аута, рассмотрим следующий пример кода:

```
setTimeout(function repeatMe(){
  /* Длинный блок кода... */
  setTimeout(repeatMe, 10);
}, 10);
setInterval(() => {
  /* Длинный блок кода... */
}, 10);
```

Установить тайм-аут, который будет повторно активизироваться через каждые 10 мс

Установить интервальный таймер, который будет срабатывать через 10 мс

Оба приведенных выше фрагмента кода могут показаться функционально эквивалентными, но на самом деле они таковыми не являются. В частности, код функции `setTimeout()` будет всегда выполняться с задержкой не меньше 10 мс после выполнения предыдущего обратного вызова (в зависимости от состояния очереди событий эта задержка может быть больше, но никогда не меньше). А код, передаваемый функции `setInterval()`, будет по возможности запускаться через каждые 10 мс, независимо от того, когда именно была выполнена последняя функция обратного вызова. И, как было показано в преды-

дущем примере, функции обработки событий от интервального таймера могут запускаться друг за другом независимо от величины задержки.

Как мы уже знаем, точное время запуска функций обратного вызова для тайм-аутов по истечении времени задержки не гарантируется. Другими словами, они не будут запускаться через каждые 10 мс, как это происходит в случае интервальных таймеров, а только планировать себя к запуску с задержкой 10 мс после того, как будет вызвана функция обратного вызова.

Все это очень важно знать. В частности, зная, каким образом интерпретатор JavaScript обрабатывает асинхронный код, особенно при большом количестве асинхронных событий, которые обычно происходят на типичной вебстранице, можно заложить прочное основание для грамотного написания фрагментов прикладного кода.

А теперь, разобравшись в особенностях работы таймеров и цикла ожидания событий, выясним, чем полученные о них знания могут нам помочь в преодолении некоторых препятствий, связанных с производительностью.

13.2.2. Преодоление трудностей затратной обработки вычислений

Вероятно, самым трудным для преодоления скрытым препятствием при разработке сложного приложения на JavaScript является однопоточный характер программирования на этом языке. Вследствие этого взаимодействие выполняемого кода JavaScript с пользователем в лучшем случае замедляется, а в худшем — код вообще перестает реагировать на действия пользователя. В итоге браузер зависает и все обновления воспроизводимой страницы задерживаются на время выполнения кода JavaScript.

В силу этого обстоятельства разделение всех сложных операций продолжительностью более нескольких сотен миллисекунд на поддающиеся управлению части становится насущной потребностью. Кроме того, большинство браузеров выводят диалоговое окно, предупреждающее пользователя, что сценарий перестал реагировать, если он выполнялся безостановочно не менее 5 секунд, тогда как другие браузеры без всяких предупреждений удаляют любые сценарии, выполняющиеся больше 5 секунд.

Когда родня встречается по какому-нибудь поводу, среди собравшихся нередко находится особенно разговорчивый родственник, который без умолку рассказывает всем одни и те же семейные истории. И если кто-нибудь не остановит его и не ввернет вовремя словечко, то общий разговор перестанет быть приятным для всех, разумеется, кроме этого словоохотливого родственника. Аналогичная ситуация возникает и с кодом, выполнение которого занимает все отведенное время, в результате чего интерфейс приложения слабо реагирует на действия пользователя, что, естественно, никуда не годится. Тем не менее не исключены и такие ситуации, в которых приходится обрабатывать большие массивы данных, как, например, при манипулировании тысячами элементов модели DOM.

И в подобных случаях особенно полезными оказываются таймеры. Ведь они способны эффективно задерживать выполнение фрагмента кода JavaScript до более подходящего момента, а также разбивать отдельные фрагменты кода на мелкие части, выполнение которых не должно привести к зависанию браузера. Принимая все это во внимание, можно преобразовать интенсивно выполняющиеся циклы и вычислительные операции в ряд не блокирующих браузер задач. В качестве примера рассмотрим код из листинга 13.4, где выполнение задачи, скорее всего, займет немало времени.

Листинг 13.4. Длительно выполняющаяся задача

```
<table><tbody></tbody></table>
<script>
  const tbody = document.querySelector("tbody");
  for (let i = 0; i < 20000; i++) {
    const tr = document.createElement("tr");
    for (let t = 0; t < 6; t++) {
      const td = document.createElement("td");
      td.appendChild(document.createTextNode(i + "," + t));
      tr.appendChild(td);
    }
    tbody.appendChild(tr);
  }
</script>
```

В данном примере кода генерируется в общем 240 тыс. узлов модели DOM, создающих таблицу с большим числом ячеек. Это невероятно затратная по вычислениям операция, которая, скорее всего, приведет к зависанию браузера и препятствованию нормальному взаимодействию его с пользователем. Ее можно сравнить со словоохотливым родственником, которого только и слышно в общем разговоре на семейном мероприятии.

В подобной ситуации требуется каким-то образом заставить разговорчивого родственника замолчать на некоторое время, чтобы дать другим родственникам возможность подключиться к разговору. По аналогии в код вводятся таймеры, чтобы прервать выполнение длительной задачи, как демонстрируется в примере кода из листинга 13.5.

Листинг 13.5. Прерывание длительной задачи с помощью таймера

```
const rowCount = 20000;          // Подготовить исходные данные
const divideInto = 4;
const chunkSize = rowCount/divideInto;
let iteration = 0;
const table = document.getElementsByTagName("tbody")[0];
setTimeout(function generateRows(){
  const base = chunkSize * iteration;   // Продолжить вычисление с прежнего места
  for (let i = 0; i < chunkSize; i++) {
    const tr = document.createElement("tr");
```

```
    for (let t = 0; t < 6; t++) {
      const td = document.createElement("td");
      td.appendChild(
         document.createTextNode((i + base) + "," + t +
                                  "," + iteration));
      tr.appendChild(td);
    }
    table.appendChild(tr);
  }
  iteration++;
  if (iteration < divideInto)
    setTimeout(generateRows, 0);
},0);
```

Запланировать следующую стадию вычислений

Установить нулевую выдержку времени, чтобы указать, что следующий шаг цикла следует выполнить "как можно скорее", но после обновления пользовательского интерфейса

В данном примере видоизмененного кода длительная операция разбита на четыре более мелкие операции, в каждой их которых создается своя доля узлов модели DOM. Эти операции с намного меньшей вероятностью приведут к прерыванию нормальной работы браузера, как показано на рис. 13.14.

Рис. 13.14. Разбиение длительных задач на более мелкие задачи, не стопорящие цикл ожидания событий, с помощью таймеров

Рассматриваемый здесь код организован таким образом, что значения данных, управляющих всей операцией, записаны в легко изменяемых переменных (`rowCount`, `divideInto` и `chunkSize`) на тот случай, если потребуется разбить операцию, скажем, на десять частей, а не на четыре.

Кроме того, для отслеживания того места, на котором была остановлена предыдущая стадия вычислений, требуется выполнить некоторые математические расчеты (`base = chunkSize * iteration`). А следующая стадия вычислений планируется автоматически до тех пор, пока они не завершатся полностью:

```
if (iteration < divideInto)
  setTimeout(generateRows, 0);
```

Но самое замечательное, что для применения рассматриваемого здесь нового асинхронного подхода к выполнению длительного задания нам потребовалось совсем немного кода. Затратив еще немного труда, можно организовать контроль за ходом выполнения операции, чтобы воочию убедиться, что все идет как надо и создать планировщик частей. А в остальном основная часть кода мало чем отличается от первоначального варианта из предыдущего примера.

Примечание

В данном случае использована нулевая задержка времени. Если вы внимательно изучили цикл ожидания событий, то должны знать, что такая задержка времени совсем не означает, что обратный вызов будет выполнен немедленно, т.е. через 0 мс. Она, напротив, предписывает браузеру выполнить обратный вызов как можно скорее. Но, в отличие от микрозадач, в промежутках между выполнением отдельных задач ему разрешается выполнить пересчет элементов интерфейса страницы. Благодаря этому браузер может обновить пользовательский интерфейс, а веб-приложения — оперативнее реагировать на действия пользователей.

С точки зрения удобств для пользователя рассматриваемый здесь подход к выполнению длительной задачи отличается от исходного еще и тем, что продолжительное прерывание работы браузера теперь заменяется четырьмя наглядными обновлениями страницы, хотя их может быть и больше. Пытаясь выполнить более мелкие фрагменты кода как можно быстрее, браузер будет также воспроизводить изменения в модели DOM после каждого цикла работы таймера. А в первоначальном варианте рассматриваемого здесь кода требовалось ждать единого массового обновления.

Чаще всего такие обновления происходят незаметно для пользователя, но об этом все же не следует забывать. И поэтому нужно стремиться к тому, чтобы любой код, помещаемый на веб-страницу, не ощутимо прерывал нормальную работу браузера.

Просто удивительно, насколько полезным может быть рассматриваемый здесь способ. Уяснив принцип действия цикла ожидания событий, можно преодолеть ограничения, накладываемые однопоточной средой выполнения

браузера, и создать у пользователей приятное впечатление от работы веб-приложения.

А теперь, когда стали понятны роли, которые играют таймеры и цикл ожидания событий в преодолении трудностей, связанных с выполнением сложных операций, рассмотрим более подробно принцип действия самих событий.

13.3. Обработка событий

Когда происходит определенное событие, его можно обработать в прикладном коде. Как уже не раз было показано в данной книге, зарегистрировать обработчики событий можно, в частности, с помощью встроенного метода `addEventListener()`, как демонстрируется в примере кода из листинга 13.6.

Листинг 13.6. Регистрация обработчиков событий

```
<button id="myButton">Click</button>
<script>
  const myButton = document.getElementById("myButton");
  myButton.addEventListener("click", function myHandler(event){
    assert(event.target === myButton,
           "The target of the event is also myButton");
    assert(this === myButton,
           "The handler is registered on myButton");
  });
</script>
```

Зарегистрировать обработчик событий, используя метод `addEventListener()`

Получить доступ к элементу, где произошло событие, через свойство `target` переданного события

По ссылке `this` в обработчике событий происходит обращение к элементу, где этот обработчик зарегистрирован

В данном примере кода определяется элемент `myButton` экранной кнопки и регистрируется обработчик событий для этого элемента. С этой целью вызывается встроенный метод `addEventListener()`, доступный всем элементам разметки документа.

Как только наступит событие от щелчка на экранной кнопке, браузер вызовет соответствующий обработчик событий — в данном случае функцию `myHandler()`. Этому обработчику событий браузер передает объект события, содержащий свойства, из которых можно извлечь сведения о наступившем событии, включая положение курсора мыши или тип нажатой кнопки мыши, если речь идет о событии от щелчков кнопками мыши, или же сведения о нажатой клавише, если речь идет о событии от клавиатуры.

К числу свойств передаваемого объекта события относится свойство `target`, содержащее ссылку на тот элемент, где произошло событие. Прежде чем исследовать принцип действия событий дальше, подготовим для этого почву, чтобы выяснить, каким образом события распространяются по модели DOM.

Примечание

В обработчике событий, как и в большинстве других функций, можно пользоваться ключевым словом **`this`**. Обычно принято считать, что в обработчике событий ключевое слово **`this`** ссылается на тот объект, где произошло событие, Но, как

станет ясно в дальнейшем, это не совсем так. Ключевое слово `this`, напротив, ссылается на тот элемент, для которого *зарегистрирован* данный обработчик событий. Откровенно говоря, элемент, для которого зарегистрирован обработчик событий, как правило, является *именно* тем элементом, где и происходит обрабатываемое событие, хотя из этого правила имеются исключения, которые будут рассмотрены далее.

13.3.1. Распространение событий по модели DOM

Как упоминалось в главе 2, элементы разметки организованы в HTML-документах в виде дерева. У каждого элемента разметки может быть нуль или больше порожденных элементов, а также единственный родительский элемент, кроме корневого элемента разметки `html`. А теперь допустим, что имеется страница, где один элемент вложен в другой и у обоих элементов имеется обработчик событий, как демонстрируется в примере кода из листинга 13.7.

Листинг 13.7. Вложенные элементы и обработчики событий

```
<html>
  <head>
    <style>
      # outerContainer {width:100px; height:100px;
                        background-color: blue;}
      # innerContainer {width:50px; height:50px;
                        background-color: red;}
    </style>
  </head>
  <body>
    <div id="outerContainer">
      <div id="innerContainer"></div>
    </div>
    <script>
      const outerContainer = document.getElementById("outerContainer");
      const innerContainer = document.getElementById("innerContainer");
      outerContainer.addEventListener("click", () => {
        report("Outer container click");
      });

      innerContainer.addEventListener("click", () => {
        report("Inner container click");
      });

      document.addEventListener("click", () => {
        report("Document click");
      });
    </script>
  </body>
</html>
```

Создать два вложенных элемента

Зарегистрировать обработчик событий от щелчков кнопками мыши для внешнего контейнера

Зарегистрировать обработчик событий для внутреннего контейнера

Зарегистрировать обработчик событий для всего документа в целом

В данном примере кода определяются два элемента HTML-разметки, `outerContainer` и `innerContainer`, которые содержатся в глобальном элементе `document`. И для всех трех объектов регистрируется обработчик событий от щелчков кнопками мыши.

А теперь допустим, что пользователь щелкнет на элементе `innerContainer`. А поскольку элемент `innerContainer` содержится в элементе `outerContainer`, а оба эти элемента — в элементе `document`, то, очевидно, что такая организация HTML-документа должна обеспечить запуск на выполнение всех трех обработчиков событий, выводящих разные сообщения. Но не совсем очевиден порядок, в котором должны выполняться обработчики событий.

Должны ли мы следовать порядку, в котором были зарегистрированы обработчики событий? Должны ли мы начинать с элемента, где происходит событие, и двигаться вверх или же сверху вниз к целевому элементу? Раньше, на заре разработки браузеров, когда принималось это решение, два основных конкурента на рынке браузеров, компания Netscape и корпорация Microsoft, выбрали совершенно противоположные модели обработки событий.

В модели обработки событий, предложенной компанией Netscape, обработка событий начинается с верхнего элемента и следует вниз к целевому элементу, являющемуся инициатором события. В таком случае обработчики событий в рассматриваемом здесь примере кода выполнялись бы в следующем порядке: обработчик событий от щелчков на элементе разметки `document`, далее обработчик событий от щелчков на элементе разметки `outerContainer` и, наконец, обработчик событий от щелчков на элементе разметки `innerContainer`. Это так называемая стратегия *перехвата событий*.

Корпорация Microsoft решила пойти другим путем: начинать с целевого элемента и двигаться вверх по дереву модели DOM. В таком случае обработчики событий в рассматриваемом здесь примере кода выполнялись бы в следующем порядке: обработчик событий от щелчков на элементе разметки `innerContainer`, далее обработчик событий от щелчков на элементе разметки `outerContainer` и обработчик событий от щелчков на элементе разметки `document`. Это так называемая стратегия *всплывания событий*.

Обе упомянутые выше модели обработки событий приняты в стандарте, установленном консорциумом W3C (W3 Consortium; `http://www.w3.org/TR/DOM-Level-3-Events/`) и реализованном во всех современных браузерах. В соответствии с этим стандартом событие обрабатывается в два этапа:

- **Перехват.** Событие сначала регистрируется в верхнем элементе разметки документа, а затем распространяется вниз к целевому элементу.
- **Всплывание.** Как только на этапе перехвата будет достигнут целевой элемент, обработка событий переходит к этапу всплывания, на котором событие всплывает, распространяясь вверх от целевого элемента снова к верхнему элементу разметки документа.

Оба этапа наглядно показаны на рис. 13.15.

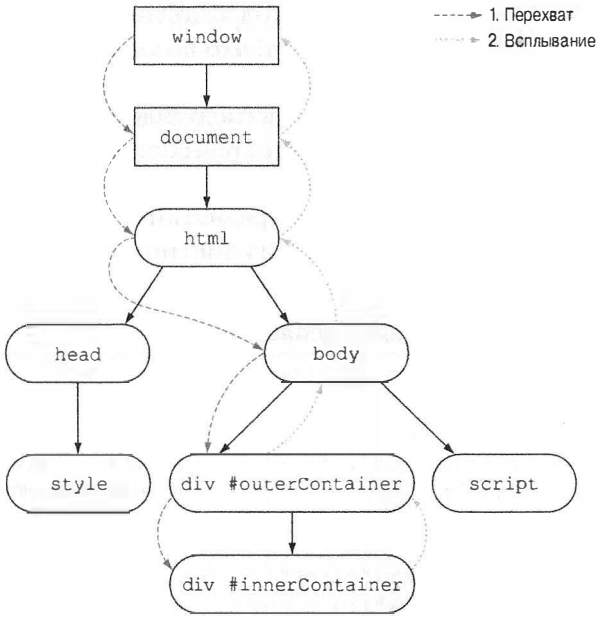


Рис. 13.15. На этапе перехвата событие распространяется вниз к целевому элементу разметки документа, а на этапе всплывания — вверх от целевого элемента

Чтобы выбрать требуемый порядок обработки событий, достаточно указать еще один, логический аргумент при вызове метода `addEventListener()`. Так, если указать логическое значение `true` в качестве третьего аргумента данного метода, то событие будет перехвачено. А если указать логическое значение `false` (или вообще опустить значение третьего аргумента), то событие будет всплывать. Очевидно, что в стандарте консорциума W3C всплыванию отдается большее предпочтение, чем перехвату событий, поскольку эта стратегия принята по умолчанию.

Вернемся к примеру кода из листинга 13.7 и выясним, каким образом регистрируются события:

```
outerContainer.addEventListener("click", () => {
  report("Outer container click");
});

innerContainer.addEventListener("click", () => {
  report("Inner container click");
});

document.addEventListener("click", () => {
  report("Document click");
});
```

Как видите, во всех трех случаях метод addEventListener() вызывается лишь с двумя аргументами, а это означает, что по умолчанию выбирается стратегия *всплывания* событий. Так, если щелкнуть на элементе innerContainer, обработчики событий будут выполнены в следующем порядке: обработчик событий от щелчков на элементе разметки innerContainer, далее обработчик событий от щелчков на элементе разметки outerContainer и, наконец, обработчик событий от щелчков на элементе разметки document.

А теперь видоизменим пример кода из листинга 13.7, как выделено полужирным в листинге 13.8.

Листинг 13.8. Перехват в сравнении со всплыванием

```
const outerContainer = document.getElementById("outerContainer");
const innerContainer = document.getElementById("innerContainer");

document.addEventListener("click", () => {
  report("Document click");
});

outerContainer.addEventListener("click", () => {
  report("Outer container click");
}, true);

innerContainer.addEventListener("click", () => {
  report("Inner container click");
}, false);
```

Если не указать третий аргумент, то по умолчанию активизируется всплывание событий

Если указать логическое значение true в качестве третьего аргумента, то активизируется перехват событий

Если же указать логическое значение false в качестве третьего аргумента, то активизируется всплывание событий

На этот раз обработчик событий в элементе outerContainer устанавливается в режим перехвата событий (путем передачи логического значения true в качестве третьего аргумента метода addEventListener()), тогда как обработчики событий в элементах innerContainer и document — в режим всплывания событий (путем передачи логического значения false в качестве третьего аргумента или опускания данного аргумента).

Как известно, единственное событие может инициировать выполнение нескольких обработчиков событий, где каждый обработчик событий может действовать в режиме перехвата или всплывания. Именно по этой причине событие проходит сначала через перехват, начиная с верхнего элемента и распространяясь вниз к целевому элементу, являющемуся инициатором данного события. Как только целевой элемент будет достигнут, активизируется режим всплывания и событие всплывает от целевого элемента в обратном направлении на самый верх.

В данном случае перехват начинается сверху, т.е. с главного объекта window, и распространяется вниз к элементу innerContainer с целью обнаружить все элементы, у которых имеется обработчик данного события в режиме перехвата. В итоге обнаруживается только один элемент outerContainer, и соответствующий обработчик от щелчков на кнопках мыши выполняется первым.

Событие продолжает распространяется вниз по пути перехвата, но больше не обнаруживается ни одного обработчика событий, действующего в режиме перехвата. Как только событие достигнет целевого элемента `innerContainer`, оно перейдет к стадии всплывания, где оно распространяется от целевого элемента в обратном направлении на самый верх, запуская попутно все обработчики событий, действующие в режиме всплывания.

В данном случае вторым выполняется обработчик событий от щелчков кнопками мыши на элементе `innerContainer`, а третьим — обработчик событий от щелчков кнопками мыши на элементе `document`. Результат щелчка кнопкой мыши на элементе `innerContainer` и путь, пройденный соответствующим событием, наглядно показаны на рис. 13.16.

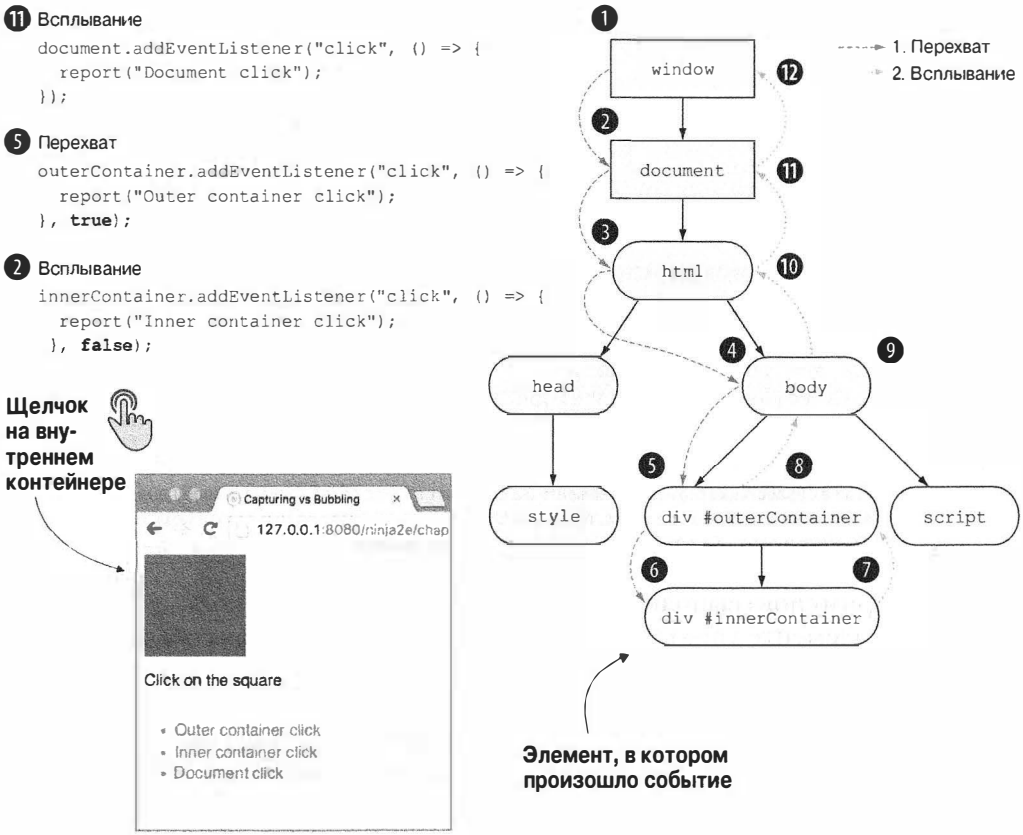


Рис. 13.16. Сначала событие распространяется сверху вниз, запуская на выполнение все обработчики событий, действующие в режиме перехвата. Как только будет достигнут целевой элемент, событие распространяется снизу вверх, запуская на выполнение все обработчики событий, действующие в режиме всплывания

В данном примере проявляется, в частности, следующая особенность: тот элемент, где обрабатывается событие, совсем не обязательно является именно тем элементом, где происходит событие. Например, в рассматриваемом здесь

примере событие происходит в элементе innerContainer, но его можно обработать в элементах, находящихся выше по иерархии в модели DOM, например, в элементе outerContainer или document.

Эта особенность возвращает нас к применению ключевого слова this в обработчиках событий и причинам, по которым явно указанное ключевое слово this обозначает ссылку на тот элемент, где зарегистрирован обработчик события, но совсем не обязательно на тот элемент, где происходит событие. Чтобы подробно исследовать данный вопрос, внесем очередные изменения в рассматриваемый здесь пример кода, как показано в листинге 13.9.

Листинг 13.9. Отличия ссылок this и event.target в обработчиках событий

```
const outerContainer = document.getElementById("outerContainer");
const innerContainer = document.getElementById("innerContainer");

innerContainer.addEventListener("click", function(event){
  report("innerContainer handler");
  assert(this === innerContainer,
         "This referes to the innerContainer");
  assert(event.target === innerContainer,
         "event.target refers to the innerContainer");
});

outerContainer.addEventListener("click", function(event){
  report("outerContainer handler");
  assert(this === outerContainer,
         "This refers to the outerContainer");
assert(event.target === innerContainer,
         "event.target refers to the innerContainer");
});
```

Обе ссылки this и event.target указывают на элемент innerContainer в обработчике событий именно этого элемента

Если в обработчике событий в элементе outerContainer обрабатывается сообщение, возникшее в элементе innerContainer, то ссылка this будет указывать на элемент outerContainer, а ссылка event.target — на элемент innerContainer

Рассмотрим снова выполнение приложения в тот момент, когда пользователь щелкает на элементе innerContainer. В обоих обработчиках событий используется стратегия всплывания событий. Об этом свидетельствует отсутствие третьего аргумента с логическим значением true в методах addEventListener(). Поэтому будет вызван обработчик событий в элементе innerContainer. В теле этого обработчика событий проверяется, ссылаются ли ключевое слово this и свойство event.target на элемент innerContainer:

```
assert(this === innerContainer,
       "This refers to the innerContainer");
assert(event.target === innerContainer,
       "event.target refers to the innerContainer");
```

Ключевое слово this ссылается на элемент innerContainer, поскольку именно для этого элемента *зарегистрирован* текущий обработчик событий.

При этом свойство `event.target` также ссылается на тот же самый элемент `innerContainer`, поскольку именно в этом элементе *произошло* событие.

Далее событие всплывает вверх к обработчику событий, зарегистрированному для элемента `outerContainer`. На этот раз ключевое слово `this` и свойство `event.target` ссылаются на разные элементы, как подтверждается ниже.

```
assert(this === outerContainer,
       "This refers to the outerContainer");
assert(event.target === innerContainer,
       "event.target refers to the innerContainer");
```

Как и предполагалось, ключевое слово `this`, с одной стороны, ссылается на элемент `outerContainer`, поскольку это именно тот элемент, для которого зарегистрирован текущий обработчик событий. А свойство `event.target`, с другой стороны, ссылается на элемент `innerContainer`, поскольку именно в этом элементе произошло событие.

Теперь, когда стало понятно, каким образом событие распространяется по дереву модели DOM и осуществляется доступ к элементу, где первоначально произошло событие, выясним, как применить полученные знания для написания кода, менее интенсивно расходующего оперативную память.

Делегирование обработки событий родительскому элементу

Предположим, что мы хотим наглядно продемонстрировать факт щелчка пользователем кнопкой мыши на ячейке таблицы. Для этого изменим фон ячейки с белого на желтый. На первый взгляд, сделать это совсем не трудно. Достаточно перебрать все ячейки таблицы и установить для каждой из них обработчик событий, изменяющий цвет фона в одноименном свойстве:

```
const cells = document.querySelectorAll('td');
for (let n = 0; n < cells.length; n++) {
  cells[n].addEventListener('click', function(){
    this.style.backgroundColor = 'yellow';
  });
}
```

Такой код вполне работоспособен, но вряд ли его можно назвать изящным. В этом коде, по существу, устанавливается один и тот же обработчик событий для элементов, которых может быть не одна сотня, и каждый раз он делает *одно и то же*.

Намного более изящный подход состоит в том, чтобы установить единственный обработчик событий на более высоком уровне, чем ячейки таблицы и обрабатывать в нем все события, используя стратегию всплывания. Как известно, все ячейки являются дочерними элементами таблицы, в которую они входят, а ссылку на элемент, на котором был выполнен щелчок кнопкой мыши, можно получить через свойство `event.target`. Было бы намного более изящно *делегировать* обработку событий таблице следующим образом:

```
const table = document.getElementById('someTable');
table.addEventListener('click', function(event){
  if (event.target.tagName.toLowerCase() === 'td')
    event.target.style.backgroundColor = 'yellow';
});
```

Выполнить действие только в том случае, если щелчок выполнен на ячейке, а не в произвольном месте таблицы

В данном случае устанавливается один обработчик событий, легко справляющийся с обязанностью изменять цвет фона в тех ячейках таблицы, на которых щелкнули кнопкой мыши. Это намного более эффективно и изящно.

Делегируя обработку события, следует убедиться, что оно применяется только к родительским элементам тех элементов, которые являются инициаторами событий. Таким образом, всплывающие события в конечном итоге достигают элемента, которому делегирован их обработчик.

13.3.2. Специальные события

Допустим, нам требуется выполнить некое действие, инициировать которое нужно при различных условиях из разных частей программы, возможно, даже из кода, находящегося в общих файлах сценариев. Начинающий программист повторил бы в этом случае код там, где это требуется. Более опытный программист создал бы глобальную функцию и организовал бы ее вызов в нужном месте. А настоящий мастер программирования на JavaScript воспользовался бы для этой цели специальными событиями. Обсудим далее основания для такого решения.

Слабое связывание

Предположим, мы хотим выполнить некоторую операцию из общего кода, причем условие для запуска этой операции должно приниматься в коде на вебстранице. Если пойти по пути создания глобальной функции, то на всех вебстраницах, где используется общий код, придется сделать так, чтобы эта функция с фиксированным именем была доступна и написать код для ее вызова, что является существенным недостатком такого подхода.

А что, если нам потребуется выполнить несколько действий при наступлении инициирующего условия? Подход, при котором создаются несколько уведомлений нельзя назвать удачным, поскольку он достаточно трудоемкий и создает совершенно ненужную путаницу в коде. Все эти недостатки являются результатом *сильного связывания*, при котором в коде, обнаруживающем условие, должны быть известны подробности о коде, который будет реагировать на это условие. С другой стороны, *слабое связывание* возникает в том случае, если в коде, инициирующем условие, ничего неизвестно о коде, который будет реагировать на это условие, или даже о самой возможности реагирования на него.

Одним из преимуществ обработчиков событий является то, что их можно установить столько, сколько потребуется, причем все они будут действовать совершенно независимо друг от друга. Именно поэтому обработка событий служит характерным примером слабого связывания. Когда возникает событие от щелчка на экранной кнопке, в коде, инициирующем это событие, ничего не-

известно о тех обработчиках событий данного типа, которые установлены на веб-странице, или даже о том, имеются ли они вообще. Вместо этого событие от щелчка на экранной кнопке помещается браузером в очередь событий, а инициатору данного события совершенно безразлично, что с ним произойдет дальше. Если для события от щелчка на экранной кнопке установлены соответствующие обработчики, то они будут в конечном итоге вызваны по отдельности и совершенно независимо друг от друга.

О слабом связывании можно было бы сказать еще немало. Что же касается рассматриваемой здесь ситуации с общим кодом, то когда в нем обнаруживается привлекающее внимание условие, формируется определенного рода сигнал, уведомляющий о следующем: произошло нечто любопытное и все, кого это заинтересует, могут отреагировать на него, а данному коду нет до того никакого дела. Обратимся к конкретному примеру.

Пример обработки Ajax-запроса

Допустим, имеется некоторый общий код, выполняющий Ajax-запрос. Веб-страницы, на которых будет использоваться этот код, должны уведомляться о том, когда этот запрос начинается и когда завершается. И на каждой веб-странице должны находиться собственные средства для реагирования на подобные события.

Например, на одной странице требуется отобразить анимационное изображение вертушки, когда начнется Ajax-запрос, а по его завершении скрыть его, чтобы дать пользователю наглядное представление о том, что данный запрос обрабатывается. Если представить себе начальное условие как событие `ajax-start`, а конечное условие — как событие `ajax-complete`, то, казалось бы, нетрудно установить на веб-странице обработчики этих событий, отображающие и скрывающие изображение вертушки в нужный момент:

```
document.addEventListener('ajax-start', e => {
  document.getElementById('whirlyThing').style.display =
           'inline-block';
});
document.addEventListener('ajax-complete', e => {
  document.getElementById('whirlyThing').style.display = 'none';
});
```

К сожалению, эти события на самом деле не существуют. Но нам ничто не мешает создать их.

Инициирование специальных событий

Специальные события — это способ имитации (с точки зрения пользователя общего кода) реального события. Но это событие должно иметь практический смысл в контексте приложения. Инициирование специального события демонстрируется в примере кода из листинга 13.10.

Листинг 13.10. Применение специальных событий

```
<style>
  #whirlyThing { display: none; }
</style>
<button type="button" id="clickMe">Start</button>
<img id="whirlyThing" src="whirly-thing.gif" />

<script>
  function triggerEvent(target, eventType, eventDetail){
    const event = new CustomEvent(eventType, {
      detail: eventDetail
    });
    target.dispatchEvent(event);
  }

  function performAjaxOperation() {
    triggerEvent(document, 'ajax-start', { url: 'my-url'});
    setTimeout(() => {
      triggerEvent(document, 'ajax-complete');
    },5000);
  }
  const button = document.getElementById('clickMe');
  button.addEventListener('click', () => {
    performAjaxOperation();
  });
  document.addEventListener('ajax-start', e => {
    document.getElementById('whirlyThing').style.display =
             'inline-block';
    assert(e.detail.url === 'my-url', 'We can pass in event data');
  });

  document.addEventListener('ajax-complete', e => {
    document.getElementById('whirlyThing').style.display = 'none';
  });
</script>
```

Экранная кнопка, после щелчка на которой имитируется Ajax-запрос

Изображение вертушки, обозначающее процесс загрузки, если оно показывается

Создать новое событие, используя конструктор объектов типа `CustomEvent`

Передать информацию объекту события через свойство `detail`

Вызвать встроенный метод `dispatchEvent()`, чтобы передать событие указанному элементу

Сымитировать Ajax-запрос с помощью таймера. Вначале выполнения инициируется событие `ajax-start`. По истечении достаточного периода времени, инициируется событие `ajax-complete`. Передать URL в качестве дополнительных данных о событии

Когда нажимается экранная кнопка, начинается Ajax-операция

Обработать событие `ajax-start`, показав вертушку

Проверить возможность доступа к дополнительным данным о событии

Обработать событие `ajax-complete`, скрыв вертушку

В приведенном выше примере кода реализован сценарий, описанный в предыдущем разделе: анимационное изображение вертушки должно показываться во время выполнения Ajax-операции. Эта операция инициируется щелчком на экранной кнопке.

Обработчик события `ajax-start`, как, впрочем, и обработчик события `ajax-complete`, устанавливается без всякой привязки к коду, в котором инициализируются оба события. В обработчиках этих специальных событий анимационное изображение вертушки показывается и скрывается соответственно, как показано ниже.

```
button.addEventListener('click', () => {
  performAjaxOperation();
```

```
});

document.addEventListener('ajax-start', e => {
  document.getElementById('whirlyThing').style.display =
           'inline-block';
  assert(e.detail.url === 'my-url', 'We can pass in event data');
});

document.addEventListener('ajax-complete', e => {
  document.getElementById('whirlyThing').style.display = 'none';
});
```

Следует иметь в виду, что всем трем обработчикам событий ничего неизвестно о существовании друг друга. В частности, обработчик событий от щелчков кнопками мыши не несет никакой ответственности за показ и сокрытие анимационного изображения вертушки. Сама же Ajax-операция имитируется в приведенном ниже фрагменте кода.

```
function performAjaxOperation() {
  triggerEvent(document, 'ajax-start', { url: 'my-url'});
  setTimeout(() => {
    triggerEvent(document, 'ajax-complete');
  }, 5000);
}
```

Приведенная выше функция инициирует событие `ajax-start` и посылает данные о событии (через свойство `url`), имитируя Ajax-запрос. Далее в рассматриваемой здесь функции делается задержка по времени на 5 секунд, чтобы сымитировать обработку Ajax-запроса в течение этого периода времени. По истечении установленной задержки по времени имитируется возврат ответа на запрос и инициируется событие `ajax-complete`, чтобы обозначить завершение Ajax-операции.

Обратите внимание на отсутствие сильных связей в данном примере. Общему коду выполнения Ajax-операции ничего неизвестно о том, что предполагается сделать в коде на веб-странице при наступлении специальных событий, и даже о том, существует ли этот код на странице вообще. А код на странице организован в виде небольших независимых друг от друга обработчиков событий. В свою очередь, коду на странице ничего неизвестно о том, что делается в общем коде. Он просто реагирует на события, которые могут наступить, а могут и не наступить.

Такая низкая степень связывания помогает сделать код модульным, упрощает его написание и заметно облегчает его отладку, если в нем обнаружатся какие-нибудь ошибки. Она упрощает также совместное использование отдельных фрагментов кода и их перенос без опасения нарушить имеющуюся связь между ними. Низкая связь дает неоспоримые преимущества, когда в прикладном коде используются специальные события, а также позволяет разрабатывать приложения в намного более выразительном и гибком стиле.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Задача в цикле ожидания событий представляет действие, выполняемое браузером. Задачи разделяются на следующие категории.
  - Макрозадачи являются отдельными, самостоятельными действиями вроде создания главного документа, обработки различных событий и внесения изменений в URL.
  - Микрозадачи являются более мелкими задачами, которые должны выполняться как можно скорее. К числу их примеров относятся обратные вызовы обещаний и видоизменения в модели DOM.
- В силу однопоточного выполнения кода задачи обрабатываются по очереди. И как только начнется выполнение задачи, ее не может прервать другая задача. Как правило, цикл ожидания событий состоит, по меньшей мере, из следующих двух очередей: макро- и микрозадач.
- Таймеры предоставляют возможность асинхронно задерживать выполнение фрагмента кода, по меньшей мере, в течение некоторого количества миллисекунд.
  - Для выполнения обратного вызова после истечения указанного периода времени служит метод `setTimeout()`.
  - Для инициализации таймера, который будет периодически выполнять обратный вызов через указанный интервал времени вплоть до отмены служит метод `setInterval()`.
  - Оба метода возвращают идентификатор таймера, с помощью которого можно отменить действие таймера, вызвав методы `clearTimeout()` и `clearInterval()`.
  - Таймеры служат для разбиения затратного по вычислениям кода на поддающиеся управлению фрагменты, не стопорящие работу браузера.
- Модель DOM представляет собой иерархическое дерево элементов, по которому обычно распространяется событие, происходящее в целевом элементе (т.е. инициаторе данного события). Имеются следующие механизмы распространения событий.
  - При перехвате события оно распространяется вниз от верхнего элемента до самого целевого элемента.
  - При всплывании события оно распространяется вверх от целевого элемента до самого верхнего элемента.
- Когда вызываются обработчики событий, браузер передает также объект события. Доступ к тому элементу, где произошло событие, осуществляется через свойство `target` этого объекта события. А в теле обработ-

чика событий ключевое слово this служит для ссылки на тот элемент, для которого зарегистрирован данный обработчик событий.

- Специальные события, создаваемые с помощью встроенного конструктора объектов типа CustomEvent и передаваемые через метод dispatchEvent(), служат для ослабления связи между различными частями приложения.

Упражнения

1. Почему так важно, чтобы постановка задач в очередь происходила за пределами цикла ожидания событий?
2. Почему так важно, чтобы каждый шаг цикла ожидания событий занимал не больше 16,7 мс?
3. Какой из перечисленных ниже результатов будет выведен на консоль через 2 секунды после начала выполнения приложения?

```
setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);
```

   а) Timeout Interval Interval Interval Interval

   б) Interval Timeout Interval Interval Interval

   в) Interval Timeout Timeout

4. Какой из перечисленных ниже результатов будет выведен на консоль через 2 секунды после начала выполнения приложения?

```
const timeoutId = setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);

clearTimeout(timeoutId);
```

   а) Interval Timeout Interval Interval Interval

   б) Interval

   в) Interval Interval Interval Interval

5. Какой из перечисленных ниже результатов будет выведен на консоль в результате выполнения приведенного ниже кода и щелчка кнопкой мыши на элементе с идентификатором inner?

```
<body>
  <div id="outer">
    <div id="inner"></div>
  </div>
  <script>
    const innerElement = document.querySelector("#inner");
    const outerElement = document.querySelector("#outer");
    const bodyElement = document.querySelector("body");

    innerElement.addEventListener("click", function(){
      console.log("Inner");
    });

    outerElement.addEventListener("click", function(){
      console.log("Outer");
    }, true);

    bodyElement.addEventListener("click", function(){
      console.log("Body");
    })
  </script>
</body>
```

а) `Inner Outer Body`

б) `Body Outer Inner`

в) `Outer Inner Body`

14 Стратегии разработки кросс-браузерного кода

В этой главе...

- Стратегии разработки повторно используемого кросс-браузерного кода на JavaScript
- Анализ характера затруднений, которые приходится разрешать
- Разумный подход к разрешению возникающих затруднений

Всякий, кому приходилось разрабатывать код сценариев для веб-страниц на JavaScript, уже через пять минут осознавал наличие целого ряда затруднений, связанных с обеспечением надежной работы кода в поддерживаемых браузерах. Эти затруднения приходится принимать во внимание на разных стадиях разработки: от элементарного решения текущих задач и до планирования на случай выпуска новых версий браузеров, учитывая все способы повторного использования кода на еще не созданных веб-страницах.

Написание кода для многих браузеров — безусловно, нетривиальная задача, которая требует увязки с избранными методологиями разработки веб-приложений, а также с наличными ресурсами для работы над проектом. Как бы нам ни хотелось, для разработки веб-страниц, которые должны идеально функционировать в каждом браузере, который когда-либо существовал, имеется и еще только появится, суровая реальность вынуждает признать, что в нашем распоряжении имеются ограниченные ресурсы. Поэтому нам приходится тщательно планировать такое применение этих ресурсов, чтобы извлечь из них наибольшую пользу.

Именно поэтому настоящая глава начинается с рекомендаций относительно выбора поддерживаемых браузеров. После этого в ней обсуждаются основные трудности, связанные с разработкой кросс-браузерного кода, а также эффективные стратегии их преодоления. Начнем со способов тщательного выбора поддерживаемых браузеров.

Знаете ли вы?

Какими способами обычно преодолеваются трудности, связанные с несогласованным поведением прикладного кода в разных браузерах?

Как лучше всего сделать свой код пригодным для чужих страниц?

В чем польза от полифиллов при написании кросс-браузерных сценариев?

14.1. Соображения по поводу кросс-браузерной разработки

Совершенствование навыков программирования на JavaScript открывает широкие горизонты, особенно теперь, когда применение JavaScript вышло за пределы браузеров и достигло серверов благодаря платформе Node.js. Но когда дело доходит до разработки браузерных, т.е. клиентских веб-приложений на JavaScript, а именно этому предмету посвящена данная книга, то рано или поздно приходится сталкиваться с самими *браузерами* и весьма неприятными вопросами их несовместимости.

В идеальном случае все браузеры должны работать безошибочно и поддерживать на постоянной основе веб-стандарты, но всем хорошо известно, что в действительности дело обстоит совсем иначе. И хотя качество браузеров за последнее время значительно улучшилось, им по-прежнему присущи программные ошибки, отсутствие необходимых интерфейсов API и характерные особенности работы, которые приходится учитывать при разработке веб-приложений. Выработать комплексную стратегию разрешения затруднений, связанных с браузерами, и хорошо знать их отличия и особенности работы не менее важно, если не важнее, чем грамотно программировать на JavaScript.

При написании для браузера приложений или библиотек на JavaScript очень важно правильно выбрать поддержку конкретных браузеров. Разумеется, хотелось бы поддерживать все имеющиеся браузеры, но ограничения, накладываемые на ресурсы разработки и тестирования, диктуют совершенно иной подход. Так как же решить, что именно и на каком уровне следует поддерживать?

Для ответа на этот вопрос мы можем позаимствовать старый подход, принятый в веб-службах Yahoo! и называемый *классификацией поддержки браузеров*. При таком подходе мы создаем матрицу поддержки браузеров, которая служит в качестве моментального снимка, отражающего важность браузера и его платформы для наших потребностей.

В таком матричном представлении целевые платформы указываются по одной оси, а браузеры — по другой. Затем в отдельных ячейках каждой комбинации браузера и платформы присваивается определенный класс (от A до F, хотя можно использовать и другую систему классификации в зависимости от конкретных потребностей). Гипотетический пример подобной матрицы приведен в табл. 14.1.

Таблица 14.1. Гипотетический пример матрицы поддержки браузеров

	Windows	Mac OS X	Linux	iOS	Android
IE 9		Отсутствует	Отсутствует	Отсутствует	Отсутствует
IE 10		Отсутствует	Отсутствует	Отсутствует	Отсутствует
IE 11		Отсутствует	Отсутствует	Отсутствует	Отсутствует
Edge		Отсутствует	Отсутствует	Отсутствует	Отсутствует
Firefox				Отсутствует	
Chrome					
Opera					
Safari			Отсутствует		Отсутствует

Обратите внимание на то, что в данной матрице заполнены не все ячейки. Присваивание классов конкретной комбинации платформы и браузера полностью зависит как от требований выполняемого проекта, так и от других важных факторов, в том числе от состава целевой аудитории. Применяя подобный подход, можно разработать систему классификации, в которой определяется важность поддержки платформы и браузера, объединив эту информацию со стоимостью такой поддержки, чтобы выявить оптимальный состав поддерживаемых браузеров.

Выбирая поддержку браузера, мы обычно обязуемся сделать следующее.

- Активно тестировать свой код в среде этого браузера с помощью избранного тестового набора.
- Исправлять программные ошибки и регрессии, связанные с этим браузером.
- Обеспечивать приемлемый уровень производительности своего кода при его выполнении в этом браузере.

В повседневной практике программирования нецелесообразно разрабатывать приложения, рассчитанные сразу на большое число платформ и браузеров, поэтому имеет смысл взвесить затраты и преимущества поддержки различных браузеров. В любом подобного рода анализе необходимо принимать во внимание многие факторы. Ниже перечислены самые главные из них.

- Ожидания и потребности целевой аудитории.
- Доля браузера на рынке.
- Затраты труда на поддержку браузера.

Первый фактор носит довольно субъективный характер и определяется в рамках конкретного проекта. С другой стороны, долю браузера на рынке можно зачастую определить с достаточной степенью точности на основании имеющейся информации. А приблизительную оценку затрат труда на поддержку каждого браузера можно прикинуть, учитывая функциональные возможности браузеров и соблюдение в них современных стандартов.

На рис. 14.1 приведен пример диаграммы, наглядно представляющей сведения об использовании браузеров (они получены с веб-сайта по адресу `http://gs.statcounter.com/` в апреле 2016 г.). Любой фрагмент повторно используемого кода JavaScript, будь то библиотека для массового потребления или собственный страничный код, следует разрабатывать для нормальной работы в как можно большем числе сред, уделив основное внимание браузерам и платформам, которые имеют особое значение для конечного пользователя. Для массово употребляемых библиотек это обширный ряд сред, а для более специфичных приложений он может быть сужен. Но, принимая ответственное решение относительно поддерживаемых сред, не забывайте следующее правило:

Качество не должно приноситься в жертву охвату поддерживаемых сред.

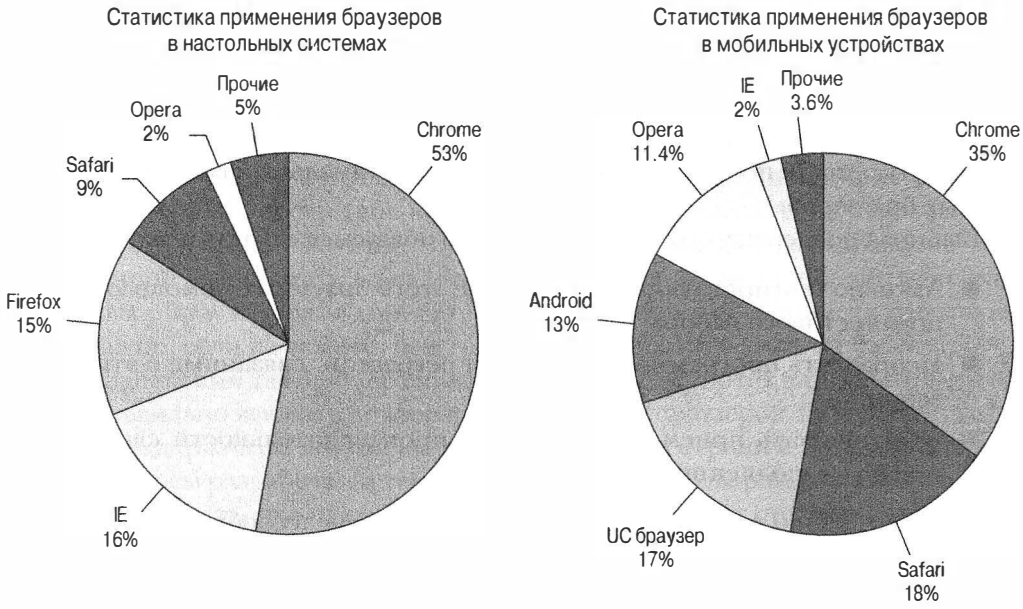


Рис. 14.1. Анализ статистики применения браузеров в настольных системах и мобильных устройствах дает ясное представление о тех браузерах, которым следует уделить основное внимание

В этой главе сначала будут рассмотрены различные ситуации, в которых код на JavaScript должен быть написан с учетом кросс-браузерной поддержки, а затем наилучшие методики написания кода с целью предотвратить любые осложнения, которые могут возникнуть в подобных ситуациях. Это поможет вам правильно выбрать те методики, принятие которых стоит затраченного времени, а следовательно, заполнить матрицу поддержки браузеров надлежащим образом.

14.2. Пять самых насущных задач разработки

При написании любого нетривиального кода возникает масса задач, которые приходится решать на стадии разработки веб-приложений. Среди них можно выделить пять самых насущных и трудных, встающих перед разработчиком при написании повторно используемого кода JavaScript (рис. 14.2).

Ниже перечислены эти пять насущных задач разработки повторно используемого кода JavaScript.

- Устранение ошибок в сценариях
- Устранение программных ошибок в браузерах.
- Отсутствующие средства в браузерах.
- Внешний код.
- Регрессии в браузерах.

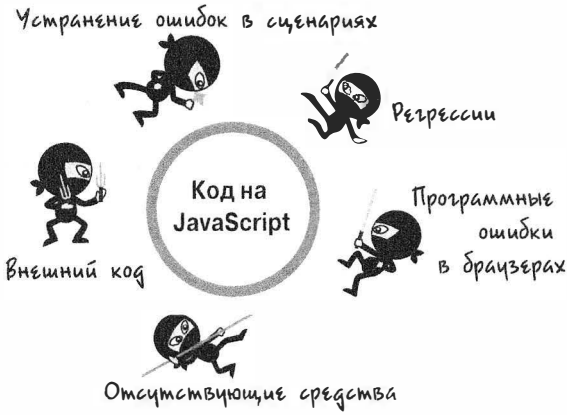


Рис. 14.2. Пять самых насущных задач разработки повторно используемого кода на JavaScript

Решая каждую из этих задач, необходимо найти золотую середину между временем, которое приходится на это тратить, и выгодой, которая будет получена в конечном итоге. В конечном счете эту золотую середину каждому разработчику приходится искать самостоятельно, исходя из сложившейся ситуации. Принимая ответственное решение, следует учитывать такие факторы, как ана-

лиз предполагаемой аудитории, наличных ресурсов для разработки и график выполнения работ.

Пытаясь разработать повторно используемый код на JavaScript, следует принимать во внимание все, что имеет отношение к данному вопросу. В первую очередь это относится к самым распространенным в настоящее время браузерам, поскольку именно ими, вероятнее всего, будет пользоваться целевая аудитория. Что же касается остальных, менее распространенных браузеров, то необходимо хотя бы убедиться, что прикладной код корректно работает с несущественной потерей некоторых функциональных возможностей. Так, если в браузере не поддерживается определенный интерфейс API, необходимо убедиться, что в проверяемом коде, по крайней мере, не генерируются исключения, препятствующие нормальной работе остального кода.

В последующих разделах перечисленные выше задачи будут рассмотрены по отдельности, чтобы вам стали понятнее возникающие трудности и пути их преодоления.

14.2.1. Программные ошибки и отличия в браузерах

Главной заботой при разработке повторно используемого кода на JavaScript должна стать обработка различных программных ошибок в браузерах и преодоление несоответствий в интерфейсе API, касающихся поддержки избранных браузеров. Это означает, что любые средства, реализованные в прикладном коде, должны быть полностью *проверены* на пригодность к применению во всех этих браузерах, несмотря на то, что возможности большинства современных браузеров в высокой степени уже унифицированы.

И достичь этой цели совсем не трудно: достаточно составить исчерпывающий набор тестов, охватывающий как типичные, так и крайние случаи применения кода. Охватив тестами разрабатываемый прикладной код в достаточной степени, мы можем быть уверены в том, что он будет надежно работать в избранном ряде поддерживаемых браузеров. А если допустить, что последующие изменения в браузерах не нарушат обратную совместимость, то можно даже надеяться на то, что код будет нормально работать и в последующих версиях этих браузеров.

Конкретные стратегии устранения программных ошибок и отличий в браузерах будут рассмотрены в разделе 14.3. Самое трудное во всем этом — реализовать устранение программных ошибок в текущих версиях браузеров таким образом, чтобы противостоять любым устранениям этих ошибок в последующих версиях браузеров.

14.2.2. Преодоление программных ошибок в браузерах

Было бы неразумно допустить, что конкретная программная ошибка будет вечно присутствовать в браузере. Ведь в большинстве браузеров программные ошибки в конечном итоге устраняются, и поэтому было бы опрометчиво рассчитывать на постоянное наличие подобных ошибок в браузерах. В этой

связи лучше всего воспользоваться методиками, рассматриваемыми в разделе 14.3, чтобы как можно более надежно застраховать любые приемы обхода программных ошибок на будущее.

При написании фрагментов повторно используемого кода на JavaScript требуется обеспечить его продолжительное время работы. Как и при разработке любого другого компонента веб-сайта (CSS, HTML и т.д.), крайне нежелательно периодически возвращаться к нему, чтобы внести изменения в веб-сайт, который не работает в новой версии какого-нибудь браузера.

Чаще всего неполадки в работе веб-сайта возникают из-за неверных допущений в отношении программных ошибок в браузерах. Они сводятся к следующему: конкретные усовершенствования вводятся с целью обойти программные ошибки, вносимые браузером, что приводит к нарушению нормальной работы веб-сайта, когда эти ошибки устраняются в последующих версиях браузера.

Вопрос обращения с программными ошибками в браузерах, по существу, распадается на два других.

1. Работоспособность разрабатываемого кода нарушится, если в браузере будет устранена ошибка.
2. Разработчики браузеров могут быть невольно приучены *не* устранять программные ошибки, опасаясь нарушить работу веб-сайтов.

Интересным примером второй ситуации служит программная ошибка, недавно обнаруженная в связи с применением свойства `scrollTop` в браузерах (`https://dev.opera.com/articles/fixing-the-scrolltop-bug/`). Обращаясь к элементам модели DOM в HTML-документе, можно воспользоваться свойствами `scrollTop` и `scrollLeft`, чтобы получить доступ к текущему положению бегунка прокрутки элемента. Но если эти свойства принадлежат корневому элементу разметки `html`, то в соответствии со спецификацией они должны сообщать (и оказывать влияние) о положении бегунка прокрутки окна просмотра. Браузеры Internet Explorer (версии IE 11) и Firefox строго следуют этой спецификации, тогда как браузеры Safari, Chrome и Opera, к сожалению, не придерживаются ее. Так, если попытаться видоизменить эти свойства в корневом элементе разметки `html`, то ничего не произойдет. Чтобы добиться того же результата в этих браузерах, придется воспользоваться свойствами `scrollTop` и `scrollLeft` элемента разметки `body`.

Когда разработчикам веб-приложений приходилось иметь дело с подобными примерами несовместимости браузеров, они нередко прибегали к следующему обходному приему: сначала выясняли текущее название браузера (по значению параметра `User Agent`, как поясняется далее), а затем вносили изменения в свойства `scrollTop` и `scrollLeft` элемента разметки `html`, если код JavaScript выполнялся в браузере Internet Explorer или Firefox, а иначе — в аналогичные свойства элемента разметки `body`, если код JavaScript выполнялся в браузере Safari, Chrome или Opera. Но, к сожалению, такой прием обхода данной программной ошибки приводил к пагубным последствиям. Ведь если явно запрограммировать внесение изменений в свойства элемента разметки

body, при условии, что код JavaScript выполняется в браузере Safari, Chrome или Opera, то данная программная ошибка на самом деле никогда так и не будет устранена в этих браузерах, поскольку ее устранение по иронии судьбы приведет к нарушениям в работе многих веб-страниц.

В связи с программными ошибками в браузерах напрашивается следующий важный вывод: если часть функциональных возможностей браузера потенциально ошибочна, следует всегда сверяться со спецификацией!

Программную ошибку в браузере необходимо отличать от неуказанной или не реализованной в спецификации интерфейса API возможности. В связи с этим целесообразно свериться со спецификациями браузеров, поскольку они содержат точные нормативы для разработки и усовершенствования кода браузеров. В то же время реализация неуказанного в спецификации программного интерфейса API может измениться в любой момент, особенно если будет предпринята попытка указать и затем внести по ходу дела изменения в эту реализацию. Что же касается несогласованности интерфейсов API, не внесенных в спецификацию, то рекомендуется всегда проверять предполагаемый результат. Следует всегда иметь в виду изменения, которые могут впоследствии произойти в этих интерфейсах API по мере того, как они примут окончательную, устоявшуюся форму.

Следует также отличать устранение программных ошибок от изменений в интерфейсе API. Если устранение программных ошибок нетрудно предвидеть (они в конечном итоге устраняются в следующих выпусках браузера, пусть даже и нескоро), то выявить изменения в интерфейсе API намного труднее. Изменения в стандартные интерфейсы API вносятся редко, хотя их нельзя полностью исключать. Намного более вероятными оказываются изменения в интерфейсах API, не внесенных в спецификацию.

Правда, такие изменения редко приводят к массовым нарушениям в работе веб-сайтов. Но если подобные нарушения все же происходят, то их, по существу, невозможно выявить заранее, если, конечно, не задаться целью проверить все интерфейсы API, когда-либо применявшиеся в разрабатываемом коде, хотя связанные с этим издержки просто неоправданны. Любые подобного рода изменения в интерфейсе API следует интерпретировать таким же образом, как и любую другую регрессию.

Что же касается следующей насущной задачи разработки повторно используемого кода на JavaScript, то следует иметь в виду, что прикладной код, как и человек, один в поле не воин. Попробуем выяснить, к каким последствиям это может привести.

14.2.3. Внешний код и разметка

Любой повторно используемый код должен сосуществовать с окружающим его кодом. Предполагается ли выполнение кода на веб-страницах, созданных собственными силами, или на веб-сайтах, разработанных другими, необходимо обеспечить нормальное его сосуществование на странице с любым другим

кодом. Но эта задача сродни палке о двух концах: код должен не только сосуществовать с неудачно написанным внешним кодом, но и не оказывать неблагоприятного воздействия на соседний код.

Решение этой насущной задачи во многом зависит от той среды, в которой предполагается использовать разрабатываемый код. Так, если повторно используемый код применяется на ограниченном количестве веб-сайтов, но рассчитан на большое число браузеров, внешний код вряд ли вызовет какие-то особые осложнения, поскольку в этом случае у разработчика имеются все необходимые полномочия и возможности самостоятельно устранять возникающие неполадки.

Совет

Данная насущная задача настолько важна, что ее рассмотрению следует посвятить отдельную книгу. Поэтому, если вас заинтересует этот предмет, рекомендуем прочитать книгу *Third-Party JavaScript* Бена Винегара и Антона Ковалева (издательство Manning Publications, 2013 г.; `https://www.manning.com/books/third-party-javascript`).

Но если попытаться разработать код для самого широкого применения, то придется каким-то образом обеспечить его надежность и эффективность. Рассмотрим некоторые стратегии для достижения этой цели.

Инкапсуляция кода

Чтобы разрабатываемый код не оказывал влияние на другие фрагменты кода на тех веб-страницах, где он загружается, его лучше всего *инкапсулировать*. Обычно под *инкапсуляцией* подразумевается "размещение внутри, как в капсуле", а в программировании: "языковой механизм для ограничения доступа к некоторым компонентам объекта". Но, попросту говоря, инкапсуляция действует по следующему принципу: не суй нос не в свое дело!

Когда код помещается на веб-страницу, он должен оставлять за собой минимальный след. На самом деле сохранить такой след совсем не трудно в нескольких глобальных переменных, а еще лучше — в одной.

Характерным тому примером может служить jQuery — самая распространенная на стороне клиента библиотека, упоминавшаяся в главе 12. В ней вводится одна глобальная переменная (функция) под названием **`jQuery`** и один псевдоним этой глобальной переменной — **`$`**. В этой библиотеке поддерживаются даже средства для возврата псевдонима **`$`** любому другому страничному коду или библиотеке, где он может быть использован.

Практически все операции в jQuery выполняются через функцию `jQuery()`. А любые другие, так называемые *служебные функции*, которые она предоставляет, определяются как ее свойства. (Как поясняется в главе 3, одни функции совсем не трудно определить как свойства других функций.) Таким образом, имя **`jQuery`** используется в качестве пространства имен для всех определений данной функции.

Подобную стратегию можно взять на вооружение. Допустим, что определяется ряд функций для собственного употребления или для применения другими и все они должны быть сгруппированы в избранном пространстве имен ninja. Как и в библиотеке jQuery, для этой цели можно определить глобальную функцию ninja(), выполняющую различные операции в зависимости от того, что именно ей передается:

```
var ninja = function(){ /* здесь следует код реализации */ }
```

Определить далее собственные служебные функции, используя эту глобальную функцию в качестве их пространства имен, не представляет большого труда, как показано ниже.

```
ninja.hitsuke = function(){
  /* здесь следует код для отвлечения огнем внимания охраны */ }
```

Если бы потребовалось, чтобы имя ninja служило не в качестве функции, а лишь пространства имен, его можно было бы определить следующим образом:

```
var ninja = {};
```

В этом случае создается пустой объект, в котором можно определить свойства и функции, чтобы не вводить их имена в глобальное пространство имен.

К числу других приемов, которых следует избегать, чтобы сохранить код инкапсулированным, относится видоизменение любых существующих переменных, прототипов функций и даже элементов модели DOM. С одной стороны, любой компонент веб-страницы, который является внешним по отношению к прикладному коду и видоизменяется им, служит потенциальным местом для конфликтов и недоразумений. А с другой стороны, даже если следовать самым лучшим методикам и тщательно инкапсулировать прикладной код, то и тогда нельзя гарантировать, что чужой код будет вести себя также безупречно.

Обращение с не менее типичным кодом

С тех пор как Грейс Хоппер удалила моль с реле на первой в мире ЭВМ Mark I, существует старая поговорка: "Единственный код, который не засасывает, — это код, написанный самостоятельно". Возможно, это и слишком циничная точка зрения, но когда прикладной код сосуществует с другим кодом, не поддающимся контролю, приходится ради перестраховки предполагать самое худшее.

Какой-нибудь код, пусть даже и грамотно написанный, а не ошибочный, может *неумышленно* выполнять такие действия, как, например, видоизменение прототипов функций, свойств объектов и методов обращения к элементам модели DOM. И какими бы благими ни были намерения авторов чужого кода, он может стать западней, в которую легко попасться.

И если в подобных случаях в нашем прикладном коде будет сделано что-нибудь совершенно безобидное, например, использованы массивы JavaScript, то никто не сможет нам поставить в вину следующее простое допущение: массивы JavaScript должны действовать именно так, как им и положено. Но если

код на какой-нибудь другой странице изменит порядок действия этих массивов, то наш прикладной код может перестать нормально работать, хотя мы и не допустили никакой оплошности.

К сожалению, существует немало устоявшихся правил поведения в подобных затруднительных ситуациях, но они не мешают предпринять ряд предупредительных мер для выхода из них. И такие меры будут представлены в последующих подразделах.

Борьба с поглощающими идентификаторами

Большинство браузеров обладают *антисвойством*, которое нельзя назвать программной ошибкой, поскольку его поведение носит совершенно намеренный характер. Оно способно нарушить нормальное выполнение прикладного кода и привести к неожиданному сбою в нем. Это антисвойство связано с поиском ссылок на другие элементы разметки с использованием значения атрибутов `id` или `name` целевого элемента. А когда значения этих атрибутов вступают в конфликт со свойствами, которые уже являются частью элемента разметки, может произойти все, что угодно.

В качестве примера рассмотрим приведенный ниже фрагмент HTML-разметки простейшей формы, чтобы выявить те неприятности, к которым могут привести так называемые *поглощающие идентификаторы*.

```
<form id="form" action="/conceal">
  <input type="text" id="action"/>
  <input type="submit" id="submit"/>
</form>
```

В браузерах эта форма вызывается следующим образом:

```
var what = document.getElementById('form').action;
```

Справедливо предположить, что переменной `what` будет присвоено значение атрибута `action` элемента разметки `form`. И в большинстве случаев так и произойдет. Но если проверить содержимое этой переменной, то обнаружится, что она, как ни странно, содержит ссылку на элемент разметки `input#action`! Проведем еще один, следующий эксперимент:

```
document.getElementById('form').submit();
```

Данный оператор должен привести к отправке данных формы на сервер, но вместо этого получается следующее сообщение об ошибке в сценарии, где извещается, что `submit()` не является функцией объекта формы:

```
Uncaught TypeError: Property 'submit' of object
        #<HTMLFormElement> is not a function
```

Так что же произошло? А все дело в том, что в браузерах ссылки на все элементы разметки формы типа `<input>` сохраняются в виде свойств элемента `<form>`. На первый взгляд это может показаться очень удобным, но по зрелом размышлении оказывается, что имена этих свойств берутся из значения атрибутов `id` или `name` элементов разметки `<input>`. А если это значение совпадет

с уже имеющимся свойством элемента формы, например `action` или `submit`, то исходные свойства будут заменены новым свойством. Такое явление называется *затиранием свойств элементов в модели DOM*.

Таким образом, перед созданием элемента разметки `input#submit` ссылка `form.action` указывает на значение атрибута `action` элемента разметки `<form>`. А впоследствии она указывает на элемент разметки `input#submit`. То же самое происходит и со ссылкой `form.submit`. Возникает двоякая ситуация. Ведь для такого поведения элементы разметки `<input>` должны содержать атрибут `id`. А если у них имеется атрибут `id`, то к ним можно без труда обращаться и не вводя свойства в форму.

Это пережиток тех времен, когда у браузеров отсутствовал богатый набор методов из интерфейса API для извлечения элементов из модели DOM. С тех пор разработчики браузеров внедрили эти средства с целью облегчить доступ к элементам формы. В настоящее время можно без особого труда получить доступ к любому элементу в модели DOM, и теперь остается лишь справиться с досадными побочными эффектами от применения этих средств.

Но в любом случае конкретное средство браузеров может вызвать немало озадачивающих осложнений в коде, поэтому его следует иметь в виду, отлаживая прикладной код в браузерах. Когда встречаются свойства, которые были необъяснимым образом преобразованы в нечто совершенно другое, чем предполагалось, причиной тому, скорее всего, является затирание свойств элементов в модели DOM.

Правда, это затруднение можно преодолеть в собственной разметке, исключив из нее простые значения атрибутов `id` и `name`, способные вступить в конфликт со стандартными именами свойств. Именно такой прием и рекомендуется применять в дальнейшем. А значения **`submit`** следует особенно избегать для атрибутов `id` и `name`, поскольку оно служит основным источником неверного и запутанного поведения кода.

Порядок загрузки таблиц стилей и сценариев

Зачастую таблица CSS-правил уже построена к тому времени, когда прикладной код начинает выполняться на веб-странице. Самый лучший способ гарантировать доступность CSS-правил, указанных в таблицах стилей, при выполнении кода JavaScript на веб-странице — указать ссылки на внешние таблицы стилей перед включением файлов внешних сценариев.

В противном случае возможны неожиданные результаты, поскольку в сценарии будет предпринята попытка получить доступ к неверной информации о стилях. К сожалению, эту проблему не так-то просто устранить только средствами JavaScript, и поэтому данную особенность следует указать в пользовательской документации.

Выше были рассмотрены лишь некоторые и самые простые примеры влияния, и зачастую совершенно неумышленного, внешних факторов на работоспособность разрабатываемого кода. Чаще всего подобные затруднения возникают при попытке пользователей внедрить разрабатываемый код на своих

сайтах. В таком случае можно организовать раннюю диагностику подобных затруднений, составив соответствующие тесты для их выявления и устранения. В других случаях подобные затруднения могут возникнуть в процессе интеграции чужого кода на собственных веб-страницах. И можно надеяться, что полезные советы, приведенные в предыдущих подразделах, помогут вам выявить причины возникающих затруднений.

К сожалению, других способов устранения осложнений в связи с подобным внедрением кода, кроме ранней диагностики и принятия предохранительных мер при написании кода, не существует. А теперь перейдем к рассмотрению следующей насущной задачи.

14.2.4. Регрессии

Регрессии представляют собой самую трудную задачу, с которой приходится иметь дело при разработке повторно используемого, устойчивого кода на JavaScript. Они представляют собой программные ошибки или обратно несовместимые изменения в интерфейсах API (главным образом, неуказанных в спецификации), вносимые браузерами и поэтому приводящие к нарушению нормальной работы кода самым непредсказуемым образом.

На заметку

Термин *регрессия* употребляется здесь в его классическом определении: работавшее раньше средство больше не функционирует должным образом. И происходит это, как правило, ненамеренно, хотя иногда обусловлено преднамеренными изменениями, нарушающими нормальную работу существующего кода.

Предвосхищение изменений

Некоторые изменения в интерфейсе API можно предвосхитить, с упреждением обнаружить и соответственно учесть, как показано в примере кода из листинга 14.1. Так, в версии 9 браузера Internet Explorer была реализована поддержка обработчиков событий модели DOM второго уровня, назначаемых с помощью метода `addEventListener()`. Поэтому в коде, написанном до появления версии IE9, данное изменение можно было легко учесть, просто проверив, существует ли данный метод у объекта.

Листинг 14.1. Предвосхищение изменений в интерфейсе API

```
function bindEvent(element, type, handle) {
   if (element.addEventListener) {
     element.addEventListener(type, handle, false);
   }
   else if (element.attachEvent) {
      element.attachEvent("on" + type, handle);
   }
}
```

Привязать к объекту, используя стандартный интерфейс API

Привязать к объекту, используя оригинальный интерфейс API

В данном примере код был написан с заделом на будущее, предвидя или хотя бы надеясь на то, что когда-нибудь браузер Internet Explorer будет приведен в соответствие со стандартами модели DOM. Если в браузере поддерживается интерфейс API, совместимый с этими стандартами, это можно легко выявить, обнаружив у объекта нужный нам метод. Если же такая поддержка отсутствует, то проверяется наличие метода, специфичного для браузера Internet Explorer, чтобы воспользоваться именно им. В противном случае ничего не делается вообще.

К сожалению, большинство предстоящих изменений в интерфейсе API или программные ошибки не так-то просто предвидеть. И это еще одно веское основание для тестирования кода, на котором постоянно делается акцент в данной книге. Перед лицом непредсказуемых изменений, способных оказать влияние на код, вся надежда остается только на прилежное тестирование кода в каждом выпуске браузера, чтобы как можно быстрее устранить затруднения, которые могут внести регрессии.

Наличие хорошего набора тестов и постоянное отслеживание появляющихся выпусков браузеров представляет собой самый лучший способ избежать подобного рода регрессий в будущем. И совсем не обязательно, что это как-то скажется на привычном цикле разработки приложений, в который уже должна входить стадия тестирования кода. Поэтому следует учитывать выполнение тестов в новых выпусках браузеров, планируя любой цикл разработки.

Сведения о появляющихся выпусках браузеров можно получить из следующих источников:

- Microsoft Edge (наследник браузера Internet Explorer): `http://blogs.windows.com/msedgedev/`.
- Firefox: `http://ftp.mozilla.org/pub/firefox/nightly/latest-mozilla-central/`.
- WebKit (Safari): `https://webkit.org/nightly/`.
- Opera: `https://dev.opera.com/`.
- Chrome: `http://chrome.blogspot.hr/`.

Такой подход требует прилежания. Ведь заранее неизвестно, когда программные ошибки будут внесены браузером, и поэтому самое лучше — постоянно контролировать работоспособность кода, чтобы предотвратить любые кризисные ситуации, которые могут наступить.

Правда, разработчики браузеров делают немало для того, чтобы предотвратить подобного рода регрессии, а тестовые наборы из различных библиотек JavaScript интегрированы в основной тестовый набор большинства современных браузеров. Благодаря этому удается выявить непосредственное влияние появляющихся впоследствии регрессий на эти библиотеки. И хотя это и не позволяет выявить все регрессии во всех браузерах, подобное начинание служит положительным признаком прогресса, наметившегося в отношении разработ-

чиков браузеров к вопросам предотвращения многих осложнений в работе с их программными продуктами.

Итак, мы рассмотрели четыре главные трудности, возникающие при разработке повторно используемого прикладного кода на JavaScript: наличие и устранение программных ошибок в браузерах, внешний код и регрессии. Отдельного рассмотрения требует пятая трудность: отсутствие нужных средств в браузерах. Поэтому мы рассмотрим ее в следующем разделе наряду с другими стратегиями реализации, пригодными для разработки кросс-браузерных веб-приложений.

14.3. Стратегии реализации

Знать и понимать те задачи, которые придется решать при разработке кросс-браузерного кода, — это лишь полдела. Но совсем другое дело — выбрать эффективную стратегию для реализации кросс-браузерного кода. Вряд ли найдется такая стратегия, которая оказалась бы пригодной в каждой ситуации, и поэтому большинство вопросов, связанных с кодовой базой, следует решать, сочетая разные стратегии. Ниже будет рассмотрен ряд таких стратегий. И начнем мы с самой простой стратегии, не требующей особых хлопот.

14.3.1. Надежное устранение ошибок в кросс-браузерном коде

К числу самых простых и надежных способов устранения ошибок в кросс-браузерном коде относятся те, которые обладают следующими важными особенностями.

- Не оказывают никакого отрицательного влияния на работу других браузеров и не проявляют никаких побочных эффектов.
- Не прибегают ни к каким формам определения типа браузера или его функциональных возможностей.

Примеры подобного способа устранения ошибок обычно встречаются редко, но именно к такой тактике следует всегда прибегать при разработке веб-приложений. Обратимся к конкретному примеру. Ниже приведен фрагмент кода, представляющий изменение, внесенное в библиотеку jQuery для нормальной работы с браузером Internet Explorer.

```
// игнорировать отрицательные значения ширины и высоты
if ( (key == 'width' || key == 'height') && parseFloat(value) < 0 )
  value = undefined;
```

В некоторых версиях браузера Internet Explorer при установке отрицательного значения свойств высоты (`height`) или ширины (`width`) в стилевом оформлении веб-страницы генерируется исключение, тогда как все остальные браузеры просто игнорируют вводимые отрицательные значения этих свойств. Чтобы обойти данное препятствие, было выбрано решение игнорировать все отрицательные значения во *всех* браузерах. Благодаря подобному изменению

в коде удалось предотвратить генерирование исключения в браузере Internet Explorer и в то же время избежать отрицательного влияния на остальные браузеры. Такое дополнение оказалось безболезненным, а пользователи получили в свое распоряжение единообразный интерфейс API. Ведь генерирование неожиданных исключений крайне нежелательно.

Другим примером подобного решения проблем в библиотеке jQuery может служить приведенный ниже код манипулирования атрибутами.

```
if ( name == "type" &&
     elem.nodeName.toLowerCase() == "input" &&
     elem.parentNode )
throw "type attribute can't be changed";
```

Браузер Internet Explorer не позволяет манипулировать атрибутом `type` в тех элементах разметки ввода данных, которые уже являются частью модели DOM. Всякая попытка сделать это приведет к генерированию специального исключения. Поэтому в библиотеке jQuery было принято компромиссное решение: предотвратить всякие попытки манипулирования атрибутом `type` во вставляемых элементах разметки ввода данных, причем во *всех* браузерах сразу. В этом случае может быть сгенерировано исключение, но только информационное.

Благодаря такому изменению в коде библиотеки jQuery отпала потребность выявлять конкретный браузер или его функциональные возможности, поскольку оно было введено как средство унификации интерфейса API для всех браузеров. И хотя упомянутое выше действие по-прежнему вызывает исключение, тем не менее оно генерируется в единообразной для всех браузеров форме.

Данное конкретное дополнение носит довольно противоречивый характер. Ведь оно явно ограничивает применение в браузерах тех компонентов библиотеки, на которые подобная программная ошибка не оказывает влияния. Разработчики библиотеки jQuery тщательно взвесили свое решение и посчитали за лучшее сделать так, чтобы интерфейс API стал унифицированным и работал согласованно, чем давал неожиданные сбои при применении в прикладном кросс-браузерном коде. Не исключено, что с подобными ситуациями придется сталкиваться всем, кто разрабатывает собственные повторно используемые кодовые базы. В этом случае необходимо тщательно проанализировать, насколько подобная ограничительная мера подходит для потенциальной аудитории пользователей.

В отношении подобных изменений в коде не следует забывать о том, что они дают решение, вполне пригодное для всех браузеров и не требуют выявления отдельного браузера или его функциональных возможностей, а следовательно, на них не будут оказывать никакого влияния какие-либо последующие изменения. Таким образом, следует всегда стремиться к подобным оптимальным решениям, даже если они находят редкое и нечастое применение.

14.3.2. Обнаружение функциональных средств и полифиллы

Как обсуждалось ранее, *обнаружение функциональных средств* относится к одному из тех приемов, к которым наиболее часто прибегают при написании кросс-браузерного кода, поскольку этот прием прост и, в общем, довольно эффективен. Он состоит в том, чтобы выявить наличие определенного объекта или его средства, и если то или другое присутствует, то сделать допущение о наличии у него подразумеваемых функциональных возможностей. (В следующем разделе мы обсудим, как быть в тех случаях, когда подобное допущение не подтверждается.)

Чаще всего обнаружение функциональных средств применяется для выбора среди нескольких интерфейсов API, предоставляющих одинаковые функциональные возможности. Например, в главе 10 был исследован метод `find()`, доступный всем массивам и применяемый для поиска первого элемента массива, удовлетворяющего определенному условию. К сожалению, этот метод доступен только в тех браузерах, которые полностью поддерживают спецификацию ES6. Что же тогда делать, столкнувшись с браузерами, где данное средство отсутствует? Иными словами, что делать в отсутствие подобных средств в браузерах?

Ответ на эти вопросы дает *полифиллы* (polyfills) – резервный вариант для браузера. Если браузер не поддерживает конкретное функциональное средство, мы можем предоставить свою реализацию этого средства. Например, в веб-службе MDN (Mozilla Developer Network – сеть разработчиков Mozilla) предоставляются полифиллы для широкого спектра функциональных средств, реализованных в стандарте ES6. К их числу, среди прочего, относится реализация в JavaScript метода `Array.prototype.find()`, описываемая в электронной документации по адресу `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find` и приведенная в листинге 14.2.

Листинг 14.2. Полифилл для метода `Array.prototype.find()`

```
if (!Array.prototype.find) {                          // Предоставить полифилл только в том случае, если данный метод не реализован в текущем браузере
  Array.prototype.find = function(predicate) {        // Указать свою реализацию
    if (this === null) {
      throw new TypeError('find called on null or undefined');
    }
    if (typeof predicate !== 'function') {
      throw new TypeError('predicate must be a function');
    }
    var list = Object(this);
    var length = list.length >>> 0;                   // Убедиться, что длина массива не является отрицательной
    var thisArg = arguments[1];
    var value;
```

```
    for (var i = 0; i < length; i++) {
      value = list[i];
      if (predicate.call(thisArg, value, i, list)) {
        return value;
      }
    }
    return undefined;
  };
}
```

Найти первый элемент массива, удовлетворяющий заданному предикату

В данном примере кода используется метод обнаружения функциональной возможности для того, чтобы проверить, имеется ли в текущем браузере встроенная поддержка метода `find()`:

```
if (!Array.prototype.find) {
  ...
}
```

При всякой возможности следует стремиться выбирать сначала указанный в спецификации способ выполнения определенного действия. Как упоминалось ранее, такой подход помогает разрабатывать прикладной код с перспективой на будущее. Именно поэтому в рассматриваемом здесь примере полифилла никаких действий не предпринимается, если искомый метод поддерживается в браузере. А если нам приходится иметь дело с браузером, где поддержка стандарта ES6 пока еще не реализована, мы предоставляем свою реализацию данного метода.

Этот метод, по существу, довольно прост. В нем осуществляется перебор всех элементов массива с вызовом предикатной функции, где проверяется, удовлетворяет ли элемент массива заданным критериям поиска. Если он удовлетворяет заданным критериям, то возвращается из функции.

Любопытная методика представлена в следующей строке кода:

```
var length = list.length >>> 0;
```

В этой строке кода знаками **>>>** обозначается *операция сдвига вправо с заполнением нулями*, сдвигающая свой первый операнд на указанное количество битов вправо с отбрасыванием лишних битов. В данном случае операция **>>>** служит для преобразования в неотрицательное число значения свойства `length` обрабатываемого массива. Это необходимо потому, что индексы массивов в JavaScript должны быть представлены целыми числами без знака.

К числу тех важных областей, в которых применяется обнаружение функциональных средств, относится выявление средств, предоставляемых той средой браузера, где выполняется код. Это дает возможность воспользоваться подобными средствами в коде, а если этого сделать нельзя, то используется резервный вариант.

В приведенном ниже фрагменте кода демонстрируется простой пример, в котором обнаружение функциональных средств используется с целью выяснить, следует ли предоставлять полноценный или же сокращенный резервный

вариант функциональных возможностей приложения для взаимодействия с пользователем.

```
if (typeof document !== "undefined" &&
    document.addEventListener &&
    document.querySelector &&
    document.querySelectorAll) {
  // Возможностей интерфейса API
  // достаточно для построения полноценного варианта приложения
}
else {
  // предоставить резервный вариант
}
```

В данном примере кода проверяется следующее:

- загружен ли документ в браузер;
- поддерживаются ли в браузере средства для установки обработчиков событий;
- способен ли браузер искать элементы разметки по имени их селектора.

Если любая из этих проверок не даст положительного результата, то придется прибегнуть к резервному варианту. И все, что делается в резервном варианте, должно отвечать ожиданиям потребителей прикладного кода, а также предъявляемым к нему требованиям. Для реализации резервного варианта имеются следующие возможности.

- Произвести дополнительное обнаружение функциональных средств и выяснить, как предоставить сокращенный вариант взаимодействия приложения с пользователем, в котором по-прежнему используется некоторый сценарий JavaScript.
- Не выполнять сценарий JavaScript, ограничившись одной только HTML-разметкой веб-страницы.
- Направить пользователя к упрощенному варианту веб-сайта (нечто подобное делается, например, с почтой GMail в Google).

Обнаружение функциональных средств требует весьма незначительных издержек на выявление объектов и их свойств и относительно просто реализуется. Поэтому данный способ вполне пригоден для реализации самых элементарных функциональных возможностей в виде резервного варианта как на уровне интерфейса API, так и на уровне самого приложения. Он может служить в качестве надежной первой линии обороны при авторской разработке повторно используемого кода.

14.3.3. Области непроверяемых ошибок в браузерах

К сожалению, в языке JavaScript и модели DOM имеется ряд проблемных областей, которые невозможно или слишком дорого проверить. Впрочем, по-

добные ситуации возникают довольно редко. Если же они все-таки возникают, то всегда имеет смысл исследовать их причины более углубленно. В последующих подразделах обсуждаются некоторые известные области ошибок, которые практически невозможно проверить обычными средствами взаимодействия с кодом на JavaScript.

Установка обработчиков событий

К числу самых неприятных изъянов в работе браузеров относится то, что невозможно определить, был ли установлен обработчик событий для данного объекта. В частности, браузеры не позволяют никоим образом выяснить, была ли назначена какая-нибудь функция в качестве обработчика событий для отдельного элемента. В силу этого из элемента просто невозможно удалить все установленные обработчики событий, если только мы не сохранили ссылки на них при создании.

Инициирование событий

Еще один изъян заключается в том, что невозможно определить, будет ли инициировано событие. Если еще можно каким-то образом определить, поддерживаются ли в браузере средства обработки событий, то совершенно *невозможно* выяснить, будет ли событие инициировано в браузере. А это может вызвать осложнения в следующих двух случаях.

Во-первых, если сценарий загружается динамически после загрузки самой веб-страницы, то в нем может быть предпринята попытка назначить обработчик событий, чтобы дождаться момента окончания загрузки окна, когда на самом деле данное событие уже наступило некоторое время назад. А поскольку определить этот факт уже нельзя, то код будет ожидать своей очереди на выполнение до бесконечности.

И во-вторых, если в сценарии предполагается обрабатывать специальные события от браузера в качестве альтернативы. Например, браузер Internet Explorer предоставляет события типа `mouseenter` и `mouseleave`, чтобы упростить процесс определения того факта, что указатель мыши входит в пределы элемента или выходит за них. Эти события нередко служат в качестве альтернативы событиям типа `mouseover` и `mouseout`, потому что они действуют несколько более интуитивно, чем стандартные события. Но поскольку, без предварительного назначения обработчиков этих событий и ожидания соответствующих действий со стороны пользователя, нельзя определить, будут ли они инициированы, то обработка таких событий в повторно используемом коде становится проблематичной.

Влияние свойств стилевого оформления

Еще одна неприятность связана с определением того влияния, которое изменение свойств стилевого оформления (CSS) оказывает на внешний вид элементов. Целый ряд свойств стилевого оформления оказывает влияние только на визуальное представление отображаемого элемента, не меняя окружающие

его элементы или же другие свойства элемента, в том числе его цвет (color), цвет фона (backgroundColor) или непрозрачность (opacity). В силу этого нельзя определить программно, приведут ли изменения в этих свойствах стилевого оформления к желаемым последствиям. А выявить их влияние можно только при визуальной проверке отображаемой страницы.

Аварийные сбои в браузерах

Тестирование сценария, приводящее к аварийному сбою в браузере, служит еще одним источником неприятностей. Код, приводящий к аварийному сбою в браузере, особенно трудно выявить. В отличие от исключений, которые легко перехватываются и обрабатываются, такой код всегда приводит к аварийному завершению работы браузера.

Например, в прежних версиях браузера Safari (см. адресу https://bugs.jquery.com/ticket/1331) составление регулярного выражения с наборами символов Юникода аналогично приведенному ниже всегда приводило к аварийному завершению работы этого браузера.

```
new RegExp("[\\w\u0128-\uFFFF*_-]+");
```

Дело в том, что нельзя проверить работоспособность такого регулярного выражения обычными средствами имитации компонентов, поскольку это будет всегда приводить к нежелательным результатам в старой версии данного браузера. Кроме того, с программными ошибками, приводящими к аварийным сбоям, очень трудно бороться, поскольку аварийные сбои браузеров для пользователей вообще неприемлемы, хотя в этих браузерах и можно запретить выполнение проблематичных сценариев JavaScript.

Несовместимость интерфейсов API

Как пояснялось ранее, в библиотеке jQuery запрещается изменять атрибут type во всех браузерах из-за программной ошибки в браузере Internet Explorer. Такую операцию можно было бы проверить и отменить только в браузере Internet Explorer, но это привело бы к тому, что интерфейс API действовал бы по-разному в различных браузерах. В подобных случаях единственная возможность состоит в том, чтобы предоставить другое решение в обход области с трудно преодолимой программной ошибкой.

Помимо областей непроверяемых ошибок и изъянов, имеются также области, допускающие проверку, хотя произвести ее эффективно не так-то просто. Рассмотрим некоторые из этих областей.

Производительность интерфейсов API

Иногда отдельные интерфейсы API работают быстрее или медленнее в различных браузерах. Поэтому при написании надежного повторно используемого кода очень важно выбрать те прикладные интерфейсы API, которые обеспечивают приемлемую производительность. К сожалению, это не всегда очевидно. Но для эффективного анализа производительности отдельного ком-

понента ему обычно требуется предоставить большой объем данных, а сам анализ выполняется довольно долго. Поэтому нельзя организовать такой анализ в коде в тот момент, пока грузится основная страница.

Неподдающиеся проверке функциональные средства доставляют немало хлопот, ограничивая возможности для эффективного написания повторно используемого кода на JavaScript. Тем не менее можно всегда найти способ обойти эти трудные препятствия. Прежде всего, применяя альтернативные приемы или создавая интерфейсы API таким образом, чтобы разрешить подобные затруднения, можно все же разрабатывать достаточно эффективный код, несмотря на очевидные сложности.

14.4. Сокращение допущений

Написание кросс-браузерного кода — это своего рода состязание в допущениях. Но проведя тщательное исследование и применяя авторский подход к разработке, можно сократить число допущений, которые делаются в прикладном коде. Всякое допущение относительно создаваемого кода влечет за собой возникающие впоследствии осложнения.

Так, если допустить, что изъян или программная ошибка будет присутствовать в отдельном браузере постоянно, такое предположение окажется чрезмерным и опасным. Напротив, тестирование на предмет ошибок, как это делалось в примерах, представленных ранее в этой главе, оказывается намного более эффективным. При написании прикладного кода следует всегда стремиться к сокращению числа допущений, уменьшая по существу, вероятность допустить ошибку и невольно создать условия для осложнений, которые могут болезненно отозваться впоследствии.

Чаще всего при написании сценариев на JavaScript такие допущения делаются при определении типа браузера пользователя. В частности, сначала анализируется свойство браузера `userAgent` (`navigator.userAgent`), и на основании этого делается допущение относительно возможного поведения браузера. К сожалению, анализ подобных строковых значений в большинстве браузеров становится источником большого числа вносимых впоследствии ошибок. Допустить, что программная ошибка будет всегда связана с конкретным браузером, — значит накликать на себя беду.

Но у допущений имеется один существенный недостаток: их нельзя исключить полностью. В какой-то момент все равно придется допустить, что браузер будет делать именно то, что ему и полагается делать. Поэтому выявление того критического момента, когда это равновесие нарушается, остается полностью в руках разработчика. Именно здесь и проявляется отличие зрелого мастера от зеленого ученика.

В качестве примера рассмотрим еще раз код назначения обработчиков событий, который приводился ранее в этой главе:

```
function bindEvent(element, type, handle) {
  if (element.addEventListener) {
```

```
    element.addEventListener(type, handle, false);
  }
  else if (element.attachEvent) {
    element.attachEvent("on" + type, handle);
  }
}
```

Попробуйте сами определить три допущения, сделанные в данном коде. Действуйте смелее, а мы подождем.

(*Слышится мелодия "Опасная тема"...*)

Ну как, удалось? В приведенном выше коде было сделано по меньшей мере три допущения.

1. Проверяемые нами свойства на самом деле являются вызываемыми функциями.
2. Это именно те функции, которые подходят для выполнения предполагаемых действий.
3. Обе функции (или методы) являются единственными возможными средствами для установки обработчиков событий.

Первым допущением можно было бы благополучно пренебречь, введя проверки, выявляющие, действительно ли анализируемые свойства являются функциями. Но тогда разрешить два остальных допущения было бы намного труднее.

В разрабатываемом коде следует всегда решать, сколько допущений можно себе позволить, исходя из требований к проекту и с учетом целевой аудитории. Зачастую сокращение числа допущений приводит к увеличению размера и сложности кодовой базы. Ничто, конечно, не мешает сократить допущения до разумного числа, но в определенный момент придется остановиться и оценить критическим взглядом, что уже имеется, а затем двигаться дальше, отталкиваясь от этого рубежа. Ведь даже код с наименьшим числом допущений может быть по-прежнему подвержен регрессиям, вносимым браузером.

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Несмотря на то что ситуация за последнее время значительно улучшилась, браузеры, к сожалению, не лишены программных ошибок и зачастую не до конца поддерживают современные веб-стандарты.
- При разработке веб-приложений на JavaScript очень важно выбрать браузеры и платформы для их поддержки.
- Качество нельзя приносить в жертву охвату поддерживаемых сред, поскольку обеспечить поддержку всех допустимых сочетаний невозможно!
- Разработка кросс-браузерного кода предполагает учет следующих факторов.

  - **Объем исходного кода.** Размер исходного файла должен быть небольшим.
  - **Издержки на производительность.** Отрицательное влияние на производительность должно быть сведено к приемлемому уровню.
  - **Качество интерфейса API.** Предоставляемые прикладные интерфейсы API должны действовать единообразно во всех браузерах.
- Магической формулы для определения оптимального сочетания перечисленных выше факторов не существует.
- Оптимальное сочетание факторов разработки каждому разработчику приходится подбирать самостоятельно.
- Умело применяя такие специальные приемы, как обнаружение функциональных средств, можно защитить повторно используемый код от любых напастей с самых разных сторон, не принося для этого непомерные жертвы.

Упражнения

1. Что следует принять во внимание, выбирая браузеры для поддержки разрабатываемого веб-приложения?
2. Объясните осложнения, к которым приводит использование поглощающих идентификаторов.
3. Что означает обнаружение функциональных средств?
4. Что такое кросс-браузерные полифиллы?

Часть V

Приложения

Приложение А

Дополнительные средства стандарта ES6

В этом приложении...

- Шаблонные литералы
- Деструктурирование
- Усовершенствования литералов объектов

В этом приложении представлены менее значительные языковые средства, введенные в стандарт ES6 и не вписывающиеся в материал предыдущих глав. В частности, *шаблонные литералы (template literals)* допускают интерполяцию символьных строк и применение многострочных символьных строк, *деструктурирование* позволяет без особого труда извлекать данные из объектов и массивов, а *усовершенствованные литералы объектов* упрощают обращение с подобными литералами.

Шаблонные литералы

Шаблонные литералы являются новым языковым средством, введенным в стандарт ES6 и предназначенным для того, чтобы манипулировать символьными строками стало намного удобнее, чем прежде. Только подумайте, сколько раз вам приходилось писать неуклюжий код, подобный следующему:

```
const ninja = {
  name: "Yoshi",
  action: "subterfuge"
};
```

```
const concatMessage = "Name: " + ninja.name + " "
                    + "Action: " + ninja.action;
```

В данном примере кода создается символьная строка с динамически вставляемыми данными. Чтобы добиться этого, пришлось прибегнуть к громоздким операциям сцепления символьных строк. Но теперь в этом нет никакой необходимости! Начиная со стандарта ES6, того же самого результата можно добиться с помощью шаблонных литералов. В качестве примера рассмотрим код из листинга А.1.

Листинг А.1. Шаблонные литералы

```
const ninja = {
  name: "Yoshi",
  action: "subterfuge"
};

const concatMessage = "Name: " + ninja.name + " "
                    + "Action: " + ninja.action;

const templateMessage = `Name: ${ninja.name} Action: ${ninja.action}`;

assert(concatMessage === templateMessage,
       "Our messages match");
```

Создать шаблонный литерал с помощью знаков обратного апострофа. В них могут содержаться выражения JavaScript, заключенные в синтаксическую конструкцию `${}`

Как видите, в стандарт ES6 введен новый тип символьной строки, которая обозначается знаками обратного апострофа (`) и может содержать заполнители, заключаемые в синтаксическую конструкцию `${}`. В этих заполнителях можно разместить любое выражение JavaScript, в том числе простую переменную, ссылку на свойство объекта (в данном примере — `ninja.action`) и даже вызов функции.

Когда интерпретируется шаблонная строка, заполнители заменяются результатом вычисления выражения JavaScript, содержащегося в этих заполнителях. Кроме того, шаблонные литералы не ограничиваются только одной строкой, обычно заключаемой в двойные или одиночные кавычки. Но ничто не мешает сделать их многострочными, как показано в примере кода из листинга А.2.

Листинг А.2. Многострочные шаблонные литералы

```
const name = "Yoshi", action = "subterfuge";
const multilineString =
`Name: ${name}
 Yoshi: ${action}`;
```

Шаблонные литералы не ограничиваются одиночными строками

А теперь, после краткого введения в шаблонные литералы, перейдем к рассмотрению деструктурирования — еще одного средства, появившегося в стандарте ES6.

Деструктурирование

Деструктурирование позволяет без особого труда извлекать данные из объектов и массивов, используя шаблоны. Допустим, имеется объект, свойства которого требуется присвоить паре переменных, как демонстрируется в примере кода из листинга А.3.

Листинг А.3. Деструктурирование объектов

```
const ninja = { name:"Yoshi", action: "skulk", weapon: "shuriken"};

const nameOld = ninja.name;
const actionOld = ninja.action;
const weaponOld = ninja.weapon;

const {name, action, weapon} = ninja;

assert(name === "Yoshi", "Our ninja Yoshi");
assert(action === "skulk", "is skulking");
assert(weapon === "shuriken", "with a shuriken");

const {name: myName, action: myAction, weapon: myWeapon} = ninja;

assert(myName === "Yoshi", "Our ninja Yoshi");
assert(myAction === "skulk", "is skulking");
assert(myWeapon === "shuriken", "with a shuriken");
```

Раньше нужно было явно присвоить значение каждого свойства объекта отдельной переменной

Деструктурирование объекта позволяет присвоить значения сразу всех свойств объекта одноименным переменным

При необходимости можно явно указать имена переменных, которым присваиваются значения свойств объекта

Как следует из листинга А.3, деструктурирование объектов позволяет легко извлечь данные из объекта литерала сразу в несколько переменных. Рассмотрим в качестве примера следующий оператор:

```
const {name, action, weapon} = ninja;
```

В этом операторе создаются три новые переменные (`name`, `action` и `weapon`), которым присваиваются значения свойств объекта, указанного в правой части данного оператора. Это свойства `ninja.name`, `ninja.action` и `ninja.weapon` соответственно.

Если же требуется обозначить переменные иначе, чем именами свойств объекта, их можно указать явно, как показано в следующем операторе:

```
const {name: myName, action: myAction, weapon: myWeapon} = ninja;
```

В данном случае создаются три переменные, `myName`, `myAction` и `myWeapon`, которым присваиваются значения указанных свойств объекта.

Как упоминалось ранее, деструктурировать можно и массивы, поскольку они являются особым видом объектов. В качестве примера рассмотрим код из листинга А.4.

Листинг А.4. Деструктурирование массивов

```
const ninjas = ["Yoshi", "Kuma", "Hattori"];
const [firstNinja, secondNinja, thirdNinja] = ninjas;

assert(firstNinja === "Yoshi", "Yoshi is our first ninja");
assert(secondNinja === "Kuma", "Kuma the second one");
assert(thirdNinja === "Hattori", "And Hattorithe third");

const [, , third] = ninjas;
assert(third === "Hattori", "We can skip items");

const [first, ...remaining] = ninjas;
assert(first === "Yoshi", "Yoshi is again our first ninja");
assert(remaining.length === 2, "There are two remaining ninjas");
assert(remaining[0] === "Kuma", "Kuma is the first remaining ninja");
assert(remaining[1] === "Hattori", "Hattori the second remaining ninja");
```

Значения элементов массива по порядку присваиваются указанным переменным

Некоторые элементы массива могут быть опущены

Оставшиеся элементы массива могут быть зафиксированы отдельно

Деструктурирование массивов несколько отличается от деструктурирования объектов, главным образом, синтаксически. В этом случае переменные заключаются в квадратные, а не в фигурные скобки, как при деструктурировании объектов. Примером тому служит приведенная ниже строка кода, где имя первого ниндзя (`Yoshi`) присваивается переменной `firstNinja`, имя второго ниндзя (`Kuma`) — переменной `secondNinja`, а имя третьего ниндзя (`Hattori`) — переменной `thirdNinja`.

```
const [firstNinja, secondNinja, thirdNinja] = ninjas;
```

Кроме того, деструктурирование массивов допускает несколько более расширенное использование. Так, если требуется пропустить некоторые элементы массива, можно опустить имена соответствующих переменных, сохранив в то же время запятые, как показано в следующей строке кода, где имена двух первых ниндзя игнорируются, а имя третьего ниндзя (`Hattori`) присваивается переменной `third`:

```
const [, , third] = ninjas;
```

Имеется также возможность извлечь из массива только определенные элементы, присвоив оставшиеся новому массиву:

```
const [first, ...remaining] = ninjas;
```

В данной строке кода значение первого элемента массива (имя ниндзя `Yoshi`) присваивается переменной `first`, а имена остальных ниндзя (`Kuma` и `Hattori`) присваиваются новому массиву `remaining`. Как видите, в данном случае оставшиеся элементы массива обозначаются таким же образом, как и оставшиеся параметры функции (с помощью операции `...`).

Усовершенствованные литералы объектов

К числу наиболее примечательных особенностей JavaScript относится простота создания объектов с помощью их литералов. Достаточно определить пару

свойств, заключив их в фигурные скобки, и новый объект будет создан. В стандарте ES6 синтаксис литералов объектов был несколько расширен. Обратимся к конкретному примеру. Допустим, требуется создать объект `ninja` и назначить для него свойство, имя которого динамически вычисляется, исходя из значения переменной в текущей области видимости, а также метод получения текущего значения данного свойства, как демонстрируется в коде из листинга А.5.

Листинг А.5. Усовершенствованные литералы объектов

```
const name = "Yoshi";
const oldNinja = {
  name: name,
  getName: function(){
    return this.name;
  }
};

oldNinja["old" + name] = true;
assert(oldNinja.name === "Yoshi", "Yoshi here");
assert(typeof oldNinja.getName === "function", "with a method");
assert("oldYoshi" in oldNinja, "and a dynamic property");

const newNinja = {
  name,
  getName(){
    return this.name;
  },
  ["new" + name]: true
};

assert(newNinja.name === "Yoshi", "Yoshi here, again");
assert(typeof newNinja.getName === "function", "with a method");
assert("newYoshi" in newNinja, "and a dynamic property");
```

Создать свойство с таким же именем, как и у переменной в текущей области видимости, и присвоить ему значение данной переменной

Определить метод для объекта

Создать свойство, имя которого вычисляется динамически

Сокращенный синтаксис присваивания свойству значения переменной с таким же именем

Сокращенный синтаксис определения метода, не требующий указывать двоеточие и ключевое слово `function`. Круглые скобки после имени обозначают метод

Вычисляемое имя свойства

Сначала в данном примере кода создается объект `oldNinja` с использованием старого синтаксиса литералов объектов, который был до введения стандарта ES6:

```
const name = "Yoshi";
const oldNinja = {
  name: name,
  getName: function(){
    return this.name;
  }
};
oldNinja["old" + name] = true;
```

А далее используется синтаксис литералов объектов, усовершенствованный в стандарте ES6. Тот же самый результат достигается более простым и ясным способом:

```
const newNinja = {
  name,
  getName(){
    return this.name;
  },
  ["new" + name]: true
};
```

На этом завершается рассмотрение новых важных языковых средств, появившихся в стандарте ES6.

Приложение Б

Средства тестирования и отладки

В этом приложении...

- Инструментальные средства для отладки кода на JavaScript
- Методики формирования тестов
- Создание набора тестов
- Обзор наиболее употребительных сред тестирования

В этом приложении представлены основные методики, применяемые для отладки и тестирования веб-приложений, разрабатываемых на стороне клиента. Ведь если не протестировать прикладной код, то как узнать, что он работает надлежащим образом? Тестирование предоставляет средства, позволяющие убедиться, что прикладной код не просто работает, а работает *правильно*.

Кроме того, надежная методика тестирования требуется для *всего* кода, но она особенно важна в тех случаях, когда внешние факторы могут оказывать влияние на выполнение кода, а именно с *такой* ситуацией нам приходится сталкиваться при разработке кросс-браузерных веб-приложений на JavaScript. И дело не только в том, что над одной кодовой базой одновременно может работать несколько разработчиков, в результате чего образуются разорванные части программного интерфейса API (это типичные трудности, с которыми приходится иметь дело всем программистам), но и в том, что разрабатываемый код приходится проверять на совместимость со всеми поддерживаемыми браузерами.

В этом приложении рассматриваются инструментальные средства и методики отладки кода JavaScript, формирования тестов на основании результатов от-

ладки и построения тестовых наборов для надежного выполнения этих тестов. Итак, приступим.

Инструментальные средства для разработки и отладки веб-приложений

Долгое время разработке веб-приложений на JavaScript недоставало элементарной инфраструктуры отладки. Единственный способ отладить код на JavaScript состоял в том, чтобы в нужных местах программы расположить операторы `alert` для извещения о значении проверяемого выражения в коде, который ведет себя странно. Нетрудно догадаться, что такой способ существенно затрудняет отладку — вид деятельности, который вряд ли можно назвать занимательным.

Правда, в 2007 году было разработано расширение Firebug для браузера Firefox. Расширению Firebug принадлежит особое место в сердцах многих разработчиков веб-приложений, поскольку именно оно стало первым инструментальным средством, предоставляющим почти такое же удобство отладки, как и в современных интегрированных средах разработки вроде Visual Studio или Eclipse. Кроме того, расширение Firebug побудило к разработке аналогичных инструментальных средств для всех основных браузеров: F12 Developer Tools (в составе браузеров Internet Explorer и Microsoft Edge), WebKit Inspector (в составе браузера Safari), Firefox Developer Tools (в составе браузера Firefox), а также Chrome DevTools (в составе браузеров Chrome и Opera). Рассмотрим эти инструментальные средства более подробно.

Firebug

Firebug как первое из современных инструментальных средств отладки веб-приложений доступно только для пользователей браузера Firefox, если нажать функциональную клавишу <F12> или щелкнуть правой кнопкой мыши в любом месте страницы и выбрать команду Inspect Element with Firebug (Инспектировать элемент с помощью Firebug) из контекстного меню. Расширение Firebug можно установить, перейдя в Firefox на веб-страницу по адресу `https://getfirebug.com/` и следуя приведенным на ней инструкциям (рис. Б.1).

В расширении Firebug предоставляются усовершенствованные функциональные возможности, а некоторые из них вообще стали первыми в области отладки веб-приложений. С их помощью можно, например, без особого труда исследовать текущее состояние модели DOM, перейдя на вкладку HTML (см. рис. Б.1, *вверху*) и выполнив на консоли специальный код JavaScript в контексте текущей страницы (см. рис. Б.1, *внизу*); отладить свой код на JavaScript, перейдя на вкладку Script (Сценарий), и даже проследить взаимодействия по сети, перейдя на вкладку Net (Сеть).

Firefox Developer Tools

Помимо расширения Firebug, пользователям Firefox доступны инструментальные средства Firefox Developer Tools. Как показано на рис. Б.2, внешне эти инструментальные средства разработки веб-приложений в Firefox очень похожи на расширение Firebug (помимо незначительных отличий в компоновке и названиях вкладок; например, вкладка HTML из Firebug переименована в Inspector в Firefox Developer Tools).

Инструментальные средства разработки веб-приложений в Firefox созданы разработчиками из компании Mozilla, которые выгодно воспользовались тесной связью этих средств с браузером Firefox, дополнив их рядом полезных возможностей. Так, на вкладке Performance (Профайлер) дается подробное представление о производительности веб-приложений. Кроме того, инструментальные средства разработки веб-приложений в Firefox созданы с учетом современных требований, предъявляемых к веб. Например, они поддерживают режим адаптивного веб-дизайна (Responsive Design), в котором можно исследовать внешний вид веб-приложений на экранах устройств, имеющих разный размер, что очень важно, поскольку современные пользователи обращаются к веб-приложениям не только со своих ПК, но из мобильных, планшетных устройств и даже телевизоров.

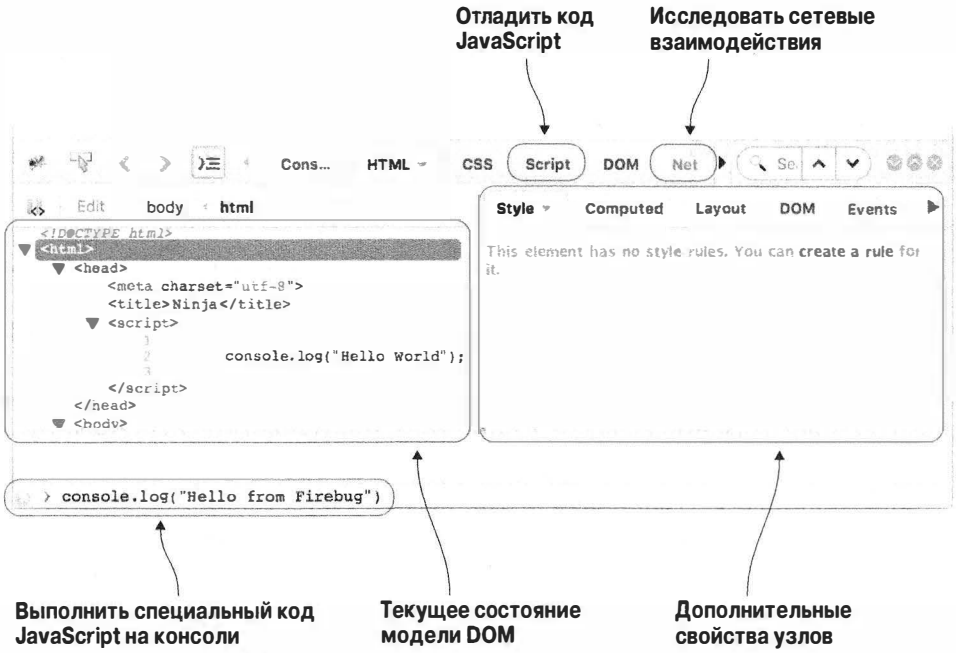


Рис. Б.1. Расширение Firefox, доступное только в браузере Firefox, стало первым современным инструментальным средством для отладки веб-приложений

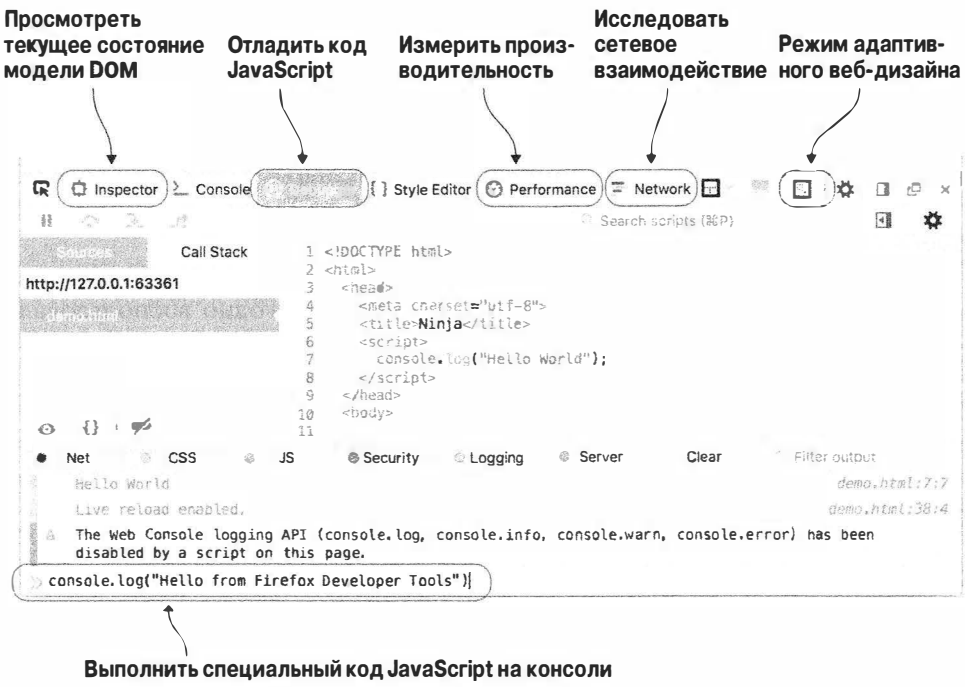


Рис. Б.2. Инструментальные средства разработки веб-приложений, встроенные в Firefox, предоставляют такие же средства отладки, как и в Firebug, и кое-что еще

F12 Developer Tools

Если вы пользуетесь браузером Internet Explorer (IE), вам, вероятно, будет приятно узнать, что он и его наследник, браузер Microsoft Edge, предоставляют свои инструментальные средства разработки веб-приложений, называемые F12 Developer Tools. (Догадайтесь, какую клавишу следует нажать, чтобы включать и выключать их.) Эти инструментальные средства приведены на рис. Б.3.

И в этом случае обращает на себя внимание сходство инструментальных средств F12 Developer Tools и Firefox Developer Tools и незначительные отличия в обозначениях названий вкладок. Кроме того, инструментальные средства F12 Developer Tools позволяют исследовать текущее состояние модели DOM (см. вкладку DOM Explorer (Проводник DOM) на рис. Б.3), выполнять специальный код JavaScript на консоли, отлаживать свой код на JavaScript (на вкладке Debugger (Отладчик)), анализировать сетевой трафик (на вкладке Network (Сеть)), работать в режиме адаптивного веб-дизайна (на вкладке UI Responsiveness (Эмуляция)), а также анализировать производительность и расход оперативной памяти (на вкладке Profiler and Memory (Производительность)).

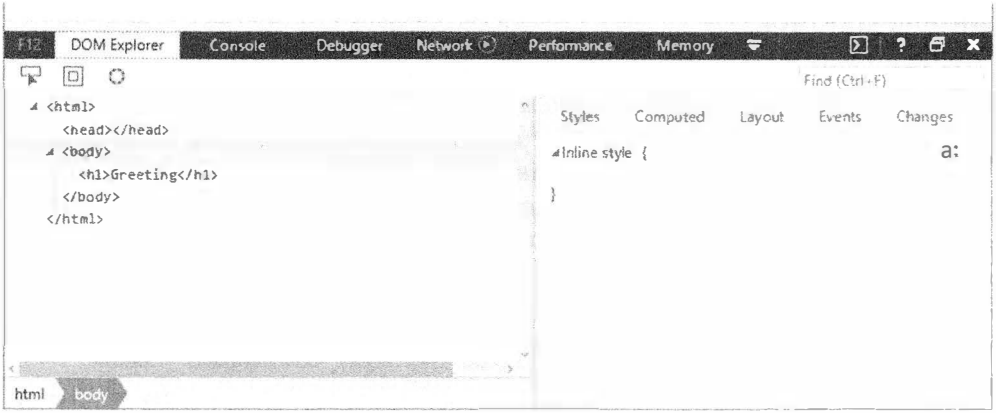


Рис. Б.3. Инструментальные средства F12 Developer Tools, включаемые и выключаемые нажатием функциональной клавиши <F12>, доступны в браузерах Internet Explorer и Edge

WebKit Inspector

Если вы работаете на платформе Mac OS X, то можете воспользоваться инструментальным средством WebKit Inspector, предоставляемым в браузере Safari (рис. Б.4). И хотя WebKit Inspector в Safari несколько отличается своим пользовательским интерфейсом от F12 Developer Tools или аналогичных инструментальных средств в Firefox, можно не сомневаться, что все основные возможности для отладки предоставляются и в WebKit Inspector.

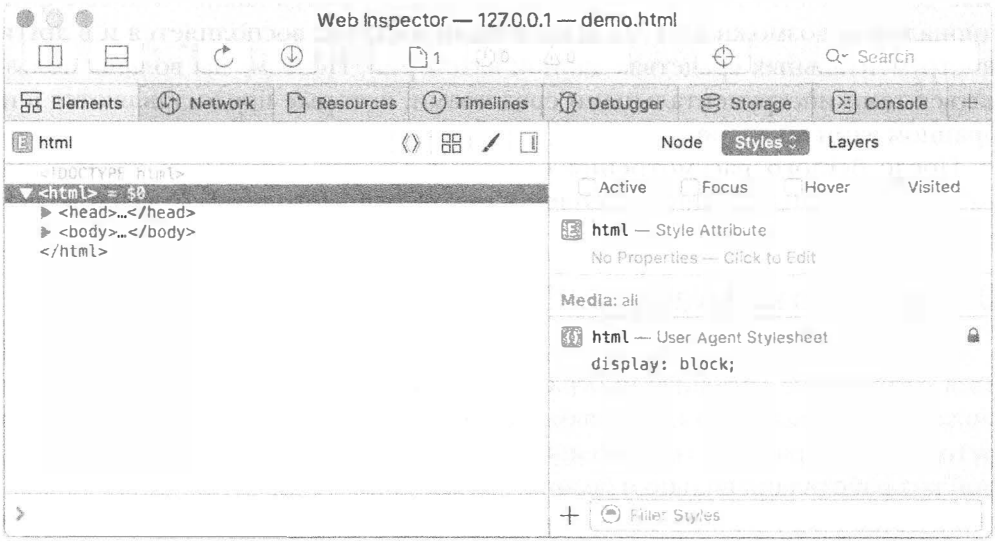


Рис. Б.4. Инструментальное средство WebKit Inspector, доступное в браузере Safari

Chrome DevTools

И в завершение краткого обзора инструментальных средств для разработки и отладки веб-приложений упомянем Chrome DevTools – самое передовое в настоящее время инструментальное средство в данной области, отличающееся немалым количеством новшеств. Как показано на рис. Б.5, его основной пользовательский интерфейс и функциональные возможности практически такие же, как и у остальных инструментальных средств данной категории.

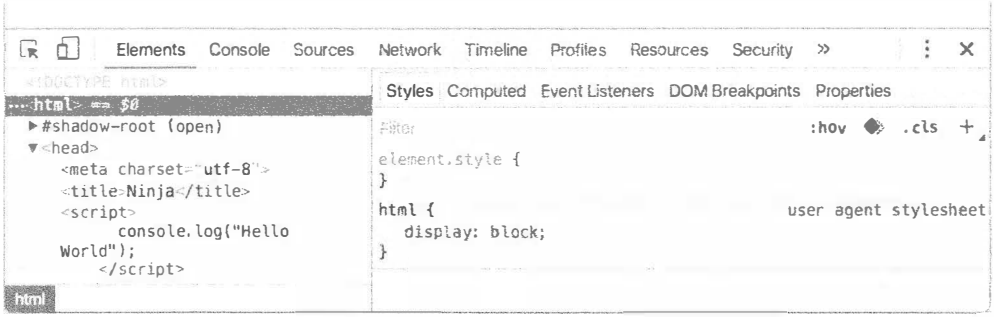


Рис. Б.5. Инструментальное средство Chrome DevTools, доступное в браузерах Chrome и Opera

Ради согласованности подачи материала при подготовке иллюстраций к данной книге было использовано инструментальное средство Chrome DevTools. Но, как будет показано далее, большинство инструментальных средств для разработки и отладки веб-приложений предоставляют аналогичные функциональные возможности. И если в одном из них появляются какие-то новые функциональные возможности, то вскоре их недостаток восполняется и в других инструментальных средствах данной категории. Поэтому вы вольны пользоваться теми инструментальными средствами, которые предоставляются в избранном вами браузере.

После беглого рассмотрения основных инструментальных средств разработки веб-приложений, перейдем к некоторым методикам отладки кода на JavaScript.

Отладка кода на JavaScript

При разработке программного обеспечения немалая доля времени уделяется устранению досадных программных ошибок. И хотя это занятие иногда может показаться занимательным подобно раскрытию тайны в детективной истории, как правило, требуется как можно скорее добиться, чтобы прикладной код работал правильно и безошибочно.

В процессе отладки кода на JavaScript применяют две основные методики, перечисленные ниже.

- **Протоколирование** – вывод информации о том, что происходит в коде при его выполнении.

- **Точки останова** — позволяют временно приостанавливать выполнение кода и анализировать текущее состояние приложения.

Обе эти методики отладки кода дают возможность найти ответ на следующий важный вопрос: что происходит в коде? Хотя каждая из них позволяет взглянуть на отлаживаемый код под разным углом зрения. Рассмотрим сначала протоколирование.

Протоколирование

Операторы *протоколирования* служат для вывода сообщений, не препятствуя обычному ходу выполнения программы. Вводя операторы протоколирования в исходный код (например, вызывая метод `console.log()`), мы можем просмотреть сообщения, выводимые на консоль браузера. Так, если требуется выяснить значение переменной x в определенные моменты выполнения программы, для этого достаточно написать код, аналогичный приведенному в примере из листинга Б.1.

Листинг Б.1. Протоколирование значения переменной x в разные моменты выполнения программы

```
<!DOCTYPE html>
1:  <html>
2:    <head>
3:      <title>Logging</title>
4:      <script>
5:        var x = 213;
6:        console.log("The value of x is: ", x);
7:
8:        x = "Hello " + "World";
9:        console.log("The value of x is now:", x);
10:     </script>
11:   </head>
12:   <body></body>
13: </html>
```

Результат выполнения данного кода в браузере Chrome, где активизирован режим работы консоли JavaScript, приведен на рис. Б.6. Как видите, протоколируемые сообщения выводятся браузером непосредственно на консоль JavaScript, где отображается как протоколируемое сообщение, так и строка кода, в которой оно протоколируется.

Это довольно простой пример протоколирования значения переменной в разные моменты выполнения программы. Но в целом с помощью протоколирования можно исследовать различные стороны выполняемых приложений, в том числе выполнение важных функций, изменение важных свойств объектов или наступление конкретных событий.

Протоколирование, безусловно, очень удобно для просмотра состояния кода при его выполнении, но иногда требуется остановить выполнение про-

граммы и проанализировать ее состояние. Именно для этой цели и служат точки останова.

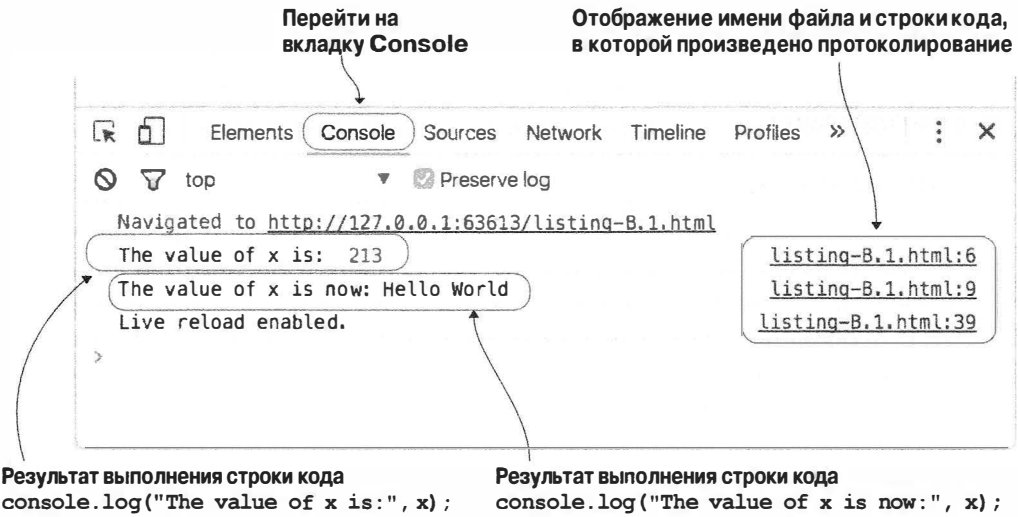


Рис. Б.6. Протоколирование дает возможность видеть состояние выполняемого кода. В данном случае значение 213 переменной **x** выводится в строке кода 6, а ее значение **"Hello World"** — в строке кода 9 из листинга Б.1. Во всех инструментальных средствах для разработки и отладки веб-приложений, включая и показанное здесь средство Chrome DevTools, для целей протоколирования предоставляется вкладка Console (Консоль)

Точки останова

Точки останова не так просты, как операторы протоколирования, но они дают заметное преимущество — они позволяют останавливать выполнение программы или сценария, а заодно и работу браузера, в конкретной строке кода. Это дает возможность тщательно проанализировать состояние всех переменных в коде в точке останова. Допустим, имеется страница, на которой выводится приветствие для известных ниндзя (выделено полужирным в примере кода из листинга Б.2).

Листинг Б.2. Простая страница с приветствиями ниндзя

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ninja greeting</title>
    <script>
      function logGreeting(name){
        console.log("Greetings to the great " + name);
      }
      var ninja = "Hattori Hanzo";
      logGreeting(ninja);
    </script>
```

Строка, где требуется остановить выполнение кода

```
  </head>
  <body>
  </body>
</html>
```

Допустим, точка останова устанавливается средствами Chrome DevTools в выделенной полужирным строке кода из листинга Б.2, где вызывается функция `logGreeting()`. Для этого достаточно перейти сначала на вкладку Debugger и щелкнуть на номере данной строки кода, а затем обновить страницу, чтобы выполнить данный код. Отладчик остановит выполнение кода на указанной строке и покажет результат, приведенный на рис. Б.7.

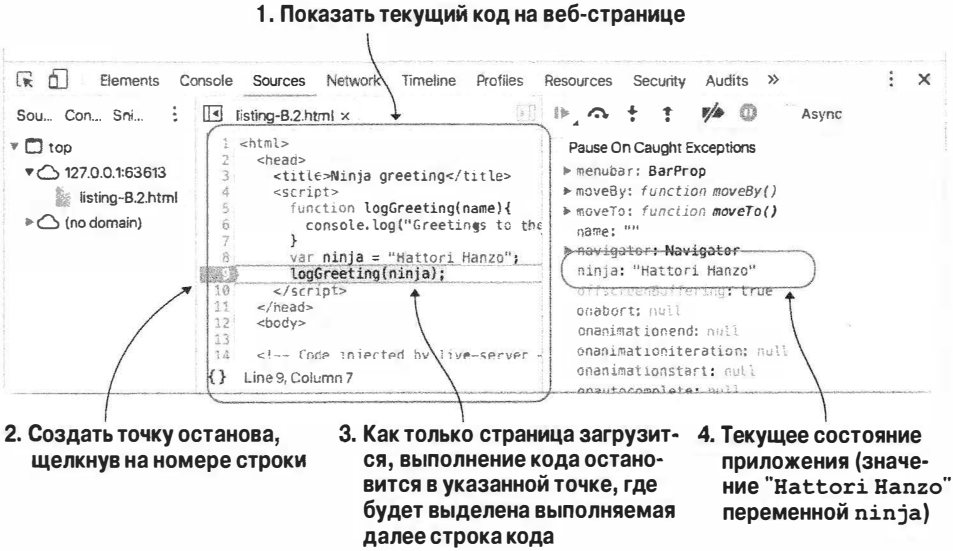


Рис. Б.7. Как только точка останова будет установлена на нужной строке кода (щелчком на ее номере), а страница загружена или обновлена, браузер остановит выполнение кода JavaScript перед тем, как выполнить указанную строку кода. Это даст возможность неспешно исследовать текущее состояние приложения на панели *справа*.

На панели справа (см. рис. Б.7) отображается текущее состояние приложения, в котором выполняется отлаживаемый код, включая значение `"Hattori Hanzo"` переменной `ninja`. Отладчик прекращает работу программы перед выполнением той строки кода, где установлена точка останова. В данном примере это строка кода, где предстоит вызвать функцию `logGreeting()`.

Заход в функцию

Если нам нужно выявить ошибку в функции `logGreeting()`, то можно *зайти* в эту функцию и посмотреть, что же в ней происходит. Как только выполнение кода остановится на операторе вызова функции `logGreeting()`, т.е. в указанной ранее точке останова, можно щелкнуть на кнопке Step Into (Зайти в),

обозначаемой стрелкой и точкой в большинстве отладчиков, или нажать функциональную клавишу <F11>. В итоге отладчик выполнит код вплоть до первой строки кода в теле функции `logGreeting()`, как показано на рис. Б.8.

Рис. Б.8. Зайдя в функцию, можно посмотреть новое состояние кода, в котором выполняется функция, а также исследовать текущее состояние стека вызовов (Call Stack) и значения локальных переменных

Обратите внимание на то, что внешний вид Chrome DevTools несколько изменился по сравнению с тем, что показано на рис. Б.7, чтобы дать возможность исследовать состояние приложения, в котором выполняется функция `logGreeting()`. Теперь можно, например, без труда проанализировать локальные переменные функции `logGreeting()` и выяснить, что переменная `name` принимает значение `"Hattori Hanzo"` (значения переменных также отображаются справа в строке кода). Обратите также внимание на панель Call Stack (Стек вызовов) справа вверху, где указывается наше положение в настоящий момент в теле функции `logGreeting()`, вызываемой из глобального кода.

Обход и выход из функции

Помимо команды Step Into, можно воспользоваться командами Step Over (Шаг с обходом) и Step Out (Шаг с выходом). В частности, команда Step Over выполняет код построчно. Если код в выполняемой строке содержит вызов функции, то отладчик обходит тело функции (сама функция будет выполняться, но обработчик не войдет в ее код).

Если же выполнение функции остановлено, то, щелкнув на кнопке Step Out, можно выполнить код до конца функции, и отладчик снова остановит выполнение программы сразу же после выхода из функции.

Точки останова по условию

Стандартные точки останова вынуждают отладчик останавливать выполнение программы всякий раз, когда он достигает конкретной точки в программе. Но иногда такая отладка превращается в слишком утомительное занятие. В качестве примера рассмотрим код из листинга Б.3.

Листинг Б.3. Подсчет ниндзя и точки останова по условию

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      for(var i = 0; i < 100; i++){
        console.log("Ninjas: " + i);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

Что, если требуется исследовать состояние приложения при подсчете 50-го ниндзя? Нужно ли для этого томительно ждать подсчета первых 49 ниндзя?

Допустим, требуется проанализировать состояние приложения при подсчете 50-го ниндзя. Насколько утомительным окажется перебор первых 49 ниндзя, чтобы достичь именно того, кто требуется?

И тут нам приходят на помощь точки останова по условию! В отличие от традиционных точек останова, которые прерывают выполнение кода всякий раз, когда будет достигнута строка, где они установлены, *точки останова по условию* вынуждают отладчик прерывать выполнение кода только в том случае, если удовлетворяется условие в выражении, связанном с этими точками останова. Чтобы установить точку останова по условию, достаточно щелкнуть правой кнопкой мыши на номере нужной строки кода и выбрать команду Add (Ввести) из контекстного меню (на рис. Б.9 показано, как это делается в браузере Chrome).

Если с точкой останова по условию связано выражение `i == 49`, отладчик остановит выполнение кода только в том случае, если будет удовлетворено условие, заданное в выражении. Это дает возможность сразу же перейти к представляющему интерес моменту выполнения кода в приложении, пренебрегая другими, менее интересными моментами.

Итак, мы показали, как пользоваться различными инструментальными средствами в разных браузерах для отладки прикладного кода путем протоколирования и с помощью точек останова. Безусловно, все эти инструментальные средства служат отличным подспорьем в выявлении программных ошибок и

достижении лучшего понимания особенностей выполнения конкретного приложения. Но помимо этого, требуется определенная среда, где можно было бы довольно рано выявлять программные ошибки. И такую среду можно организовать благодаря тестированию.

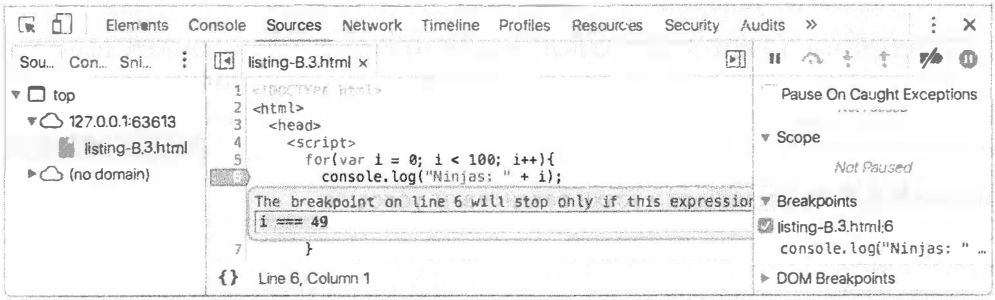


Рис. Б.9. Чтобы установить точку останова по условию, достаточно щелкнуть правой кнопкой мыши на номере нужной строки кода и выбрать команду Add из контекстного меню. Обратите внимание, что эти точки останова обозначаются другим цветом — как правило, оранжевым

Создание тестов

Известный американский поэт Роберт Фрост писал, что если изгородь хороша, то и соседи окажутся хорошими. По аналогии с этим в разработке веб-приложений и любой области программирования вообще хорошие тесты способствуют написанию хорошего кода. Подчеркнем особое значение слова *хорошие*. Ведь вполне возможно создать обширный тестовый набор, который на самом деле ничем не помогает написанию качественного кода, если тесты разработаны неудачно.

Качественные тесты обладают следующими важными свойствами.

- **Повторяемость.** Результаты тестирования должны легко воспроизводиться. Повторно выполняемые тесты должны всегда давать одни и те же результаты. Если результаты тестирования не поддаются определению, то как отличить достоверные результаты от недостоверных? Кроме того, повторяемость тестов гарантирует, что они не зависят от таких внешних факторов, как нагрузка на сеть или ЦП.
- **Простота.** Тесты должны быть нацелены на что-нибудь одно. При тестировании следует максимально упростить HTML-код, его стилевое оформление с помощью таблиц CSS или JavaScript-кода, но при этом не должен нарушать исходный контрольный пример. Чем больше элементов исключается из теста, тем больше вероятность того, что на контрольный пример будет оказывать влияние только конкретный тестируемый код.
- **Независимость.** Тесты должны выполняться обособленно. Результаты выполнения одного теста не должны зависеть от другого. Тесты следует

разделять на как можно более мелкие блоки или модули, что помогает точнее определить источник программной ошибки, когда она возникает.

В целом тесты можно разделить на две основные разновидности: *деконструктивные контрольные примеры* и *конструктивные контрольные примеры*.

- **Деконструктивные контрольные примеры.** Создаются для того, чтобы свести существующий код (путем деконструирования) к отдельной проблеме, исключая все, что к ней не относится. Благодаря этому удается достичь перечисленных выше свойств тестов. Начав с сайта в целом и последовательно исключая лишнюю разметку, стилевое оформление с помощью таблиц CSS и JavaScript-кода, можно в конечном итоге прийти к уменьшенному варианту контрольного примера, воспроизводящему искомую ошибку.
- **Конструктивные контрольные примеры.** Начинаются с известного, упрощенного примера, сложность которого наращивается (путем конструирования) до тех пор, пока не будет воспроизведена искомая программная ошибка. Для такого тестирования требуется пара простых тестовых файлов, на основании которых строятся новые тесты, а также способ формирования этих тестов из чистового варианта проверяемого кода.

Рассмотрим пример конструктивного тестирования. Создание упрощенных контрольных примеров в конечном итоге приводит к появлению нескольких HTML-файлов, в которые уже включены минимальные функциональные возможности. Для различного функционального назначения могут даже иметься разные начальные файлы: один – для манипулирования моделью DOM, другой – для тестирования средств Ajax, третий – для анимации и т.д. Так, в листинге Б.4 приведен простой контрольный пример модели DOM, используемый для тестирования библиотеки jQuery.

Листинг Б.4. Упрощенный контрольный пример модели DOM для тестирования библиотеки jQuery

```
<style>
  #test { width: 100px; height: 100px; background: red; }
</style>
<div id="test"></div>
<script src="dist/jquery.js"></script>
<script>
  $(document).ready(function() {
    $("#test").append("test");
  });
</script>
```

С другой стороны, можно воспользоваться готовой службой, предназначенной для создания простых контрольных примеров, например, JSFiddle (`http://jsfiddle.net/`), CodePen (`http://codepen.io/`) или JS Bin (`http://jsbin.`

com/?html,output). Все эти службы обладают сходными функциональными возможностями для создания контрольных примеров, которые становятся доступными по особому URL. Имеется даже возможность включать в такие примеры копии некоторых наиболее распространенных библиотек JavaScript. Пример сеанса работы со службой JSFiddle приведен на рис. Б.10.

Пользоваться службой JSFiddle (или аналогичными инструментальными средствами) удобно и практично, когда требуется быстро проверить какой-нибудь замысел, поделиться им с другими и даже получить их отзывы. К сожалению, для этого тесты придется запускать и проверять их результаты вручную. Это вполне приемлемо, если таких тестов только два, но, как правило, тестов для тщательной проверки прикладного кода оказывается намного больше. Именно поэтому тесты приходится автоматизировать как можно в большей степени. Выясним, как этого добиться на практике.

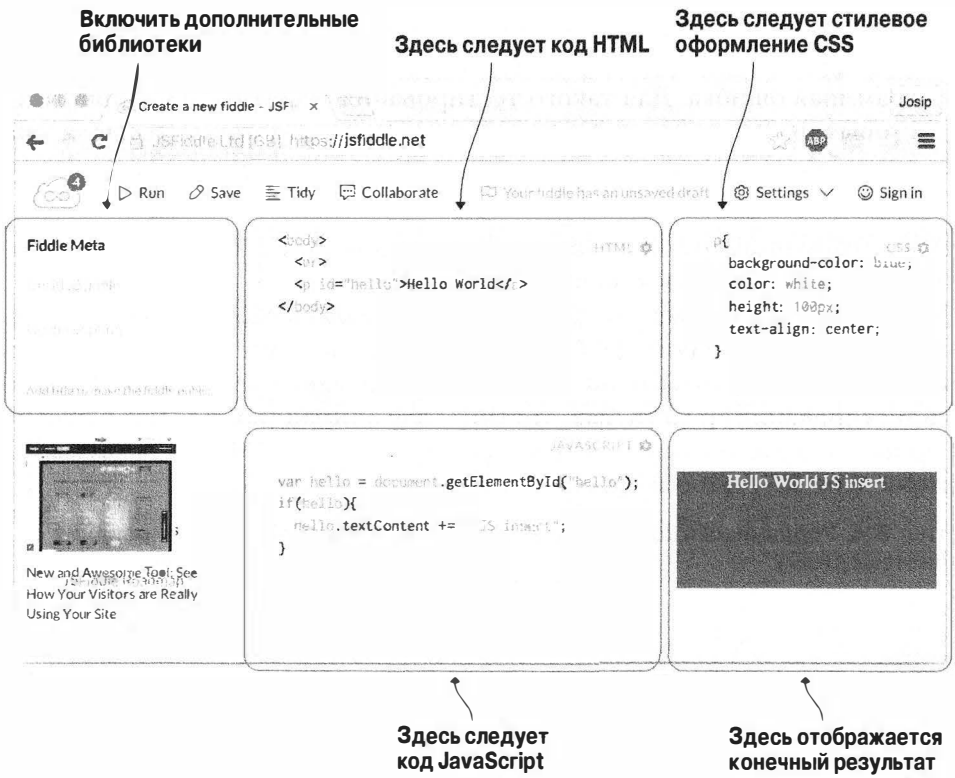


Рис. Б.10. Служба JSFiddle позволяет тестировать в различных сочетаниях HTML-разметку, стилевое оформление CSS и фрагменты кода JavaScript в "песочнице", чтобы выяснить, работает ли все именно так, как и предполагалось

Основы организации среды тестирования

Главное назначение среды тестирования — дать возможность пользователю разрабатывать отдельные тесты и включать их в единый тестовый набор, чтобы затем можно было выполнять их в массовом порядке, предоставить единый ресурс, которым можно было бы пользоваться легко и неоднократно. Чтобы лучше понять принцип действия среды тестирования, целесообразно выяснить, каким образом она создается. Как ни странно, создать среду тестирования кода JavaScript совсем не трудно.

Но в связи с изложенным выше возникает следующий вопрос: зачем вообще создавать новую среду тестированию? Чаще всего создавать собственную среду тестирования на JavaScript не требуется, поскольку уже имеется немало качественных сред тестирования, как станет ясно в дальнейшем. Но в то же время создание собственной среды тестирования может послужить хорошим уроком для приобретения необходимого опыта.

Метод утверждения

В основу среды модульного тестирования положен метод утверждения, обычно называемый `assert()`. Как правило, этому методу передается *значение*, которое предположительно должно получиться (т.е. *утверждение*) в результате вычисления выражения, а также описание назначения подобного утверждения. Если вычисляется логическое значение `true`, то утверждение проходит, а иначе оно *не* проходит. При этом выводится соответствующее сообщение с признаком прохождения или непрохождения утверждения.

Простой пример реализации принципа утверждения наглядно демонстрируется в коде из листинга Б.5.

Листинг Б.5. Простой пример реализации принципа утверждения в JavaScript

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Suite</title>
    <script>
      function assert(value, desc) {
        var li = document.createElement("li");
        li.className = value ? "pass" : "fail";
        li.appendChild(document.createTextNode(desc));
        document.getElementById("results").appendChild(li);
      }
      window.onload = function() {
        assert(true, "The test suite is running.");
        assert(false, "Fail!");
      };
    </script>
    <style>
```

Определить метод `assert()`

Выполнить тесты, используя утверждения

```
      # results li.pass { color: green; }
      # results li.fail { color: red; }
    </style>
  </head>
  <body>
    <ul id="results"></ul>
  </body>
</html>
```

Определить стили для оформления результатов

Сохранить результаты тестирования

Метод `assert()` в данном примере на удивление прост. В нем создается новый элемент разметки `<li>`, содержащий описание, затем ему присваивается класс `pass` или `fail` стилевого оформления в зависимости от значения параметра утверждения (`value`) и далее новый элемент добавляется к списку элементов в теле документа.

Тестовый набор состоит из двух простейших тестов. Один из них всегда пройдет, а другой никогда не пройдет, как показано ниже.

```
assert(true, "The test suite is running."); // пройдет всегда
assert(false, "Fail!"); // никогда не пройдет
```

Правила стилевого оформления для классов `pass` и `fail` наглядно показывают прохождение или непрохождение тестов соответствующим цветом. В книге также используется функция `report()`, которая облегчает вывод сообщений на экран. Эту функцию можно легко создать на основе функции `assert()` как показано ниже:

```
function report(text) {
    assert(true, text);
}
```

Результат выполнения рассматриваемого здесь тестового набора в браузере Chrome приведен на рис. Б.11.

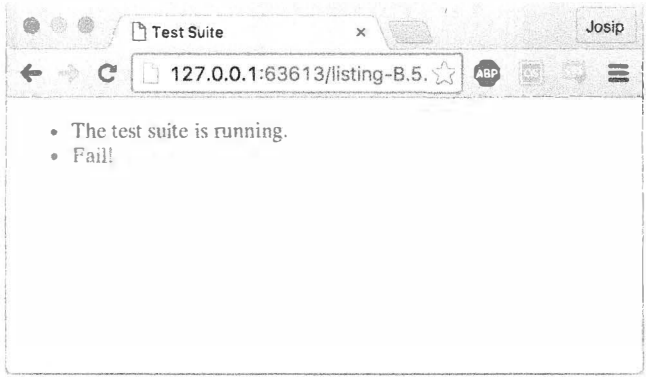


Рис. Б.11. Результат выполнения первого составленного нами тестового набора

А теперь, создав собственную элементарную среду тестирования, перейдем к представлению более совершенных и широко доступных сред тестирования.

Совет

Если для тестирования требуется более простое средство, воспользуйтесь встроенным методом `console.assert()`, как показано на рис. Б.12.

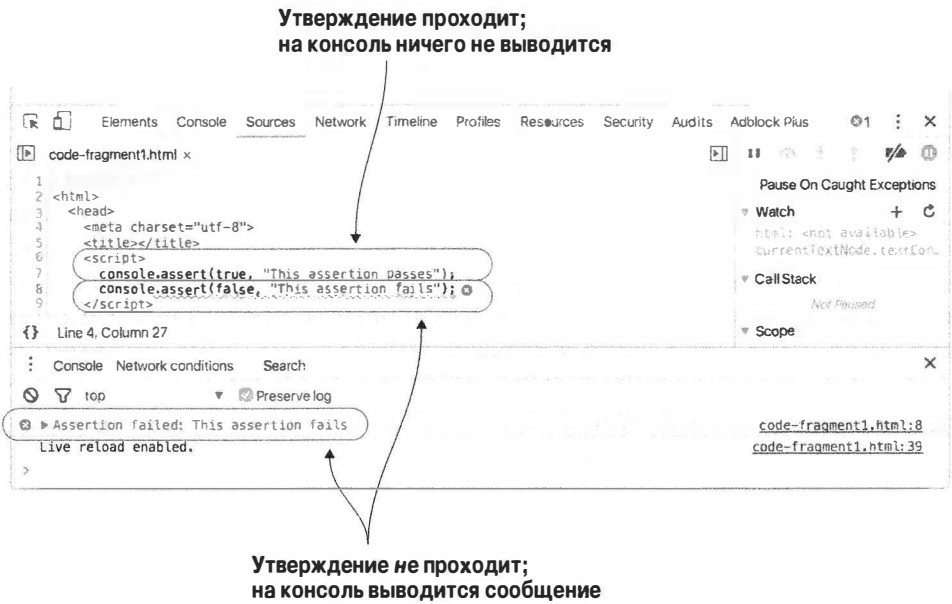


Рис. Б.12. Для оперативного тестирования прикладного кода можно воспользоваться встроенным методом `console.assert()`. Этот метод выводит на консоль сообщение только в том случае, если заданное в нем утверждение не проходит.

Наиболее распространенные среды тестирования

Среда тестирования должна служить основополагающей частью процесса разработки, поэтому очень важно выбрать такую среду тестирования, которая лучше всего подходит для вашего стиля программирования и кодовой базы. Среда тестирования кода на JavaScript должна служить единственной цели: отображать результаты выполнения тестов, чтобы упростить отделение тестов, которые успешно прошли, от тех, что не прошли. Для достижения этой цели все что от нас требуется —создать тесты и организовать их в коллекции, называемые *тестовыми наборами*. Все остальное берет на себя среда тестирования.

Существует целый ряд свойств, которыми должна обладать подходящая среда модульного тестирования, написанная на JavaScript. К их числу относятся следующие.

- Имитация поведения браузера (его реакции на действия мышью, нажатия клавиш и т.д.).

- Управление тестами в диалоговом режиме (приостановка и возобновление тестов).
- Управление асинхронными тестами по истечении времени ожидания.
- Фильтрация тестов для их выполнения.

Рассмотрим вкратце QUnit и Jasmine – две самые распространенные среды модульного тестирования.

QUnit

QUnit – это среда модульного тестирования, первоначально созданная для тестирования библиотеки jQuery. Но с тех пор ее функциональные возможности значительно расширились, и теперь она является автономной средой модульного тестирования. Среда QUnit служит, главным образом, в качестве простого решения задачи модульного тестирования, предоставляя минимальный и простой в употреблении интерфейс API.

Ниже перечислены отличительные особенности QUnit.

- Простой интерфейс API.
- Поддержка асинхронного тестирования.
- Модульное тестирование, не ограничивающееся только библиотекой jQuery и используемым ее кодом.
- Особая пригодность для регрессионного тестирования.

Рассмотрим пример теста из листинга Б.6, выполняемого в среде QUnit. В этом тесте проверяется, отвечает ли разработанная нами функция приветствием конкретному ниндзя.

Листинг Б.6. Пример теста, выполняемого в среде QUnit

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="qunit/qunit-git.css"/>
    <script src="qunit/qunit-git.js"></script>
  </head>
  <body>
    <div id="qunit"></div>
    <script>
      function sayHiToNinja(ninja) {
        return "Hi " + ninja;
      }

      QUnit.test("Ninja hello test", function(assert){
        assert.ok( sayHiToNinja("Hatori") == "Hi Hatori", "Passed");
        assert.ok( false, "Failed");
      });
    </script>
```

Включить код и стили оформления QUnit

Создать элемент HTML-разметки, который заполняется результатами тестирования в среде QUnit

Объявить функцию, которую требуется протестировать

Указать контрольный пример для тестирования в среде QUnit

Проверить прохождение утверждения

Проверить непрохождение утверждения

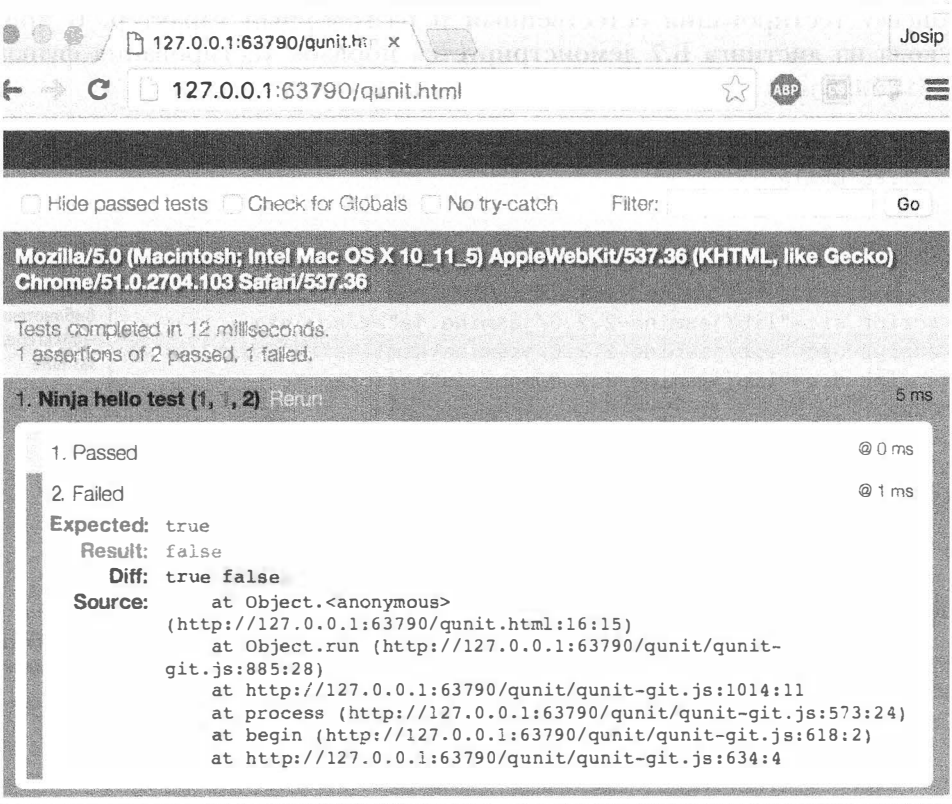
```
/body>
tml>
```

Если выполнить код из данного примера теста в браузере, то должны бы
лучены результаты, приведенные на рис. Б.13. Одно утверждение в этом
проходит при выполнении строки кода:

```
ert.ok(sayHiToNinja("Hatori") == "Hi Hatori", "Passed");
```

ругое — не проходит при выполнении следующей строки кода:

```
ert.ok(false, "Failed")
```



с. Б.13. Пример выполнения теста в среде QUnit. В этом тесте одно утверждение проходи
другое *не* проходит. В отображаемых результатах немало внимания уделяется именно то
ерждению, которое не прошло в данном тесте, чтобы можно было оперативно устрани
программную ошибку

Дополнительные сведения о среде QUnit можно найти по адресу `http:`
`nitjs.com`.

Jasmine

Еще одной распространенной для тестирования является среда Jasmine, созданная на несколько иной, чем среда QUnit, основе. Ниже перечислены основные составляющие этой среды тестирования.

- Функция `describe()`, описывающая тестовые наборы.
- Функция `it()`, определяющая отдельные тесты.
- Функция `expect()`, проверяющая отдельные утверждения.

Весь набор этих функций и их характерные имена позволяют придать процессу тестирования естественный и разговорный характер. В примере кода из листинга Б.7 демонстрируется порядок тестирования функции `sayHiToNinja()` в среде Jasmine.

Листинг Б.7. Пример теста, выполняемого в среде Jasmine

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="lib/jasmine-2.2.0/jasmine.css">

  <script src="lib/jasmine-2.2.0/jasmine.js"></script>
  <script src="lib/jasmine-2.2.0/jasmine-html.js"></script>
  <script src="lib/jasmine-2.2.0/boot.js"></script>
</head>
<body>
<script>
  function sayHiToNinja(ninja) {
    return "Hi " + ninja;
  }

  describe("Say Hi Suite", function() {
    it("should say hi to a ninja", function() {
      expect(sayHiToNinja("Hatori")).toBe("Hi Hatori");
    });

    it("should fail", function(){
      expect(false).toBe(true);
    })
  });
</script>
</body>
</html>
```

Включить библиотечные файлы среды Jasmine

Объявить функцию, которую требуется протестировать

Определить тестовый набор под названием "Say Hi Suite"

Указать один тест для проверки заданной функции

Намеренно не пройти тест

Результат выполнения тестового набора из данного примера в среде Jasmine приведен на рис. Б.14.

Дополнительные сведения о среде Jasmine можно найти по адресу `https://jasmine.github.io/`.

Измерение покрытия кода

Трудно сказать, что именно определяет качество отдельного тестового набора. В идеальном случае должны быть протестированы все возможные пути выполнения проверяемых программ. Но, к сожалению, это невозможно, за исключением самых простых случаев, поэтому в качестве шага в верном направлении можно попытаться протестировать как можно больше кода. И мерилом, определяющим степень, до которой тестовый набор покрывает проверяемый код, служит так называемое *покрытие кода*.

Рис. Б.14. Результат выполнения тестового набора из среды Jasmine в браузере. В этом наборе определены два теста, причем один проходит, а другой *не* проходит

Так, если говорят, что тестовый набор имеет 80%-ное покрытие кода, это означает, что 80% процентов кода проверяемой программы выполняется в тестовом наборе, а 20% — не охвачено им. И хотя в результате тестирования нельзя с полной уверенностью сказать, что в 80% проверяемого кода отсутствуют программные ошибки, поскольку могли быть пропущены пути выполнения кода, приводящие к одной из таких ошибок, тем не менее, об остальных 20%

невозможно судить, была ли эта часть кода вообще выполнена. Именно поэтому мерилом для тестовых наборов должно служить покрытие кода.

Для разработки веб-приложений на JavaScript можно воспользоваться рядом библиотек, помогающих измерить покрытие кода тестовыми наборами. К числу наиболее примечательных относятся библиотеки Blanket.js (`https://github.com/alex-seville/blanket`) и Istanbul (`https://github.com/gotwarlost/istanbul`). Описание установки этих библиотек выходит за рамки данной книги, но на их веб-страницах по указанным выше адресам можно найти всю информацию, необходимую для их надлежащей установки.

Приложение В

Ответы на упражнения

Глава 2. Создание страницы в динамическом режиме

1. Каковы две стадии жизненного цикла клиентского веб-приложения?

   Ответ. Двумя стадиями жизненного цикла клиентского веб-приложения являются создание страницы и обработка событий. На стадии создания страницы ее пользовательский интерфейс формируется в процессе обработки HTML-кода и выполнения основного кода на JavaScript. Как только будет обработан последний HTML-узел, страница перейдет к стадии обработки событий, на которой обрабатываются различные события.

2. Какое главное преимущество дает применение метода `addEventListener()` для регистрации обработчиков событий по сравнению с присваиванием обработчика событий свойству отдельного элемента разметки?

   Ответ. Когда обработчики событий присваиваются отдельным свойствам элементов разметки, это, дает возможность зарегистрировать только один обработчик события. С помощью метода `addEventListener()`, а с можно зарегистрировать сколько угодно обработчиков событий.

3. Сколько событий можно обработать одновременно?

   Ответ. В основу языка JavaScript положена однопоточная модель выполнения событий, которые обрабатываются по очереди.

4. В каком порядке обрабатываются события, извлекаемые из очереди событий?

   Ответ. События обрабатываются в том порядке, в каком они возникают, т.е. по принципу "первым пришел – первым обслужен".

Глава 3. Функции высшего порядка для начинающих: определения и аргументы

1. Какие функции в приведенном ниже фрагменте кода являются функциями обратного вызова?

```
// Здесь sortAsc() является функцией обратного вызова, поскольку
// в интерпретаторе JavaScript она вызывается для сравнения
// элементов массива
numbers.sort(function sortAsc(a,b){
  return a - b;
});

// А ninja() не является функцией обратного вызова,
// поскольку она вызывается как обычная функция
function ninja(){}
ninja();

var myButton = document.getElementById("myButton");
// И handleClick() является функцией обратного вызова, поскольку
// она вызывается всякий раз, когда выполняется щелчок на
// экранной кнопке myButton
myButton.addEventListener("click", function handleClick(){
  alert("Clicked");
});
```

2. Разделите функции в приведенном ниже фрагменте кода по типам, отнеся их к объявлению функции, функциональному выражению или стрелочной функции.

```
// Это функциональное выражение, указываемое в качестве
// аргумента другой функции
numbers.sort(function sortAsc(a,b){
  return a - b;
});

// Это стрелочная функция, указываемая в качестве
// аргумента другой функции
numbers.sort((a,b) => b - a);

// Это функциональное выражение служит в качестве
// вызываемого в вызывающем выражении
(function(){})();

// Это объявление функции
function outer(){
  // объявление функции
  function inner(){}
  return inner;
}

// Это вызываемое функциональное выражение, заключенное
// в другое выражение
```

```
(function(){}());

// А это стрелочная функция, указываемая в качестве вызываемой
(()=>"Yoshi")();
```

3. Какие значения примут переменные `samurai` и `ninja` после выполнения приведенного ниже фрагмента кода?

```
// Переменная samurai принимает значение "Tomoe" выражения
// в теле стрелочной функции
var samurai = (() => "Tomoe")();
// Переменная ninja принимает значение оператора return, а
// поскольку этот оператор отсутствует, то его значение
// в данном случае не определено (undefined)
var ninja = (() => {"Yoshi"})();
```

4. Какие значения принимают параметры `a`, `b` и `c` в теле функции `test()` в двух ее вызовах, приведенных ниже?

```
function test(a, b, ...c){ /*a, b, c*/}

// a = 1; b = 2; c = [3, 4, 5]
test(1, 2, 3, 4, 5);
// a = undefined; b = undefined; c = []
test();
```

5. Какие значения примут переменные `message1` и `message2` после выполнения приведенного ниже фрагмента кода?

```
function getNinjaWieldingWeapon(ninja, weapon = "katana"){
  return ninja + " " + weapon;
}

// Переменная message1 примет значение "Yoshi katana",
// поскольку в вызове функции указан один аргумент и по
// умолчанию выбирается оружие "katana"
var message1 = getNinjaWieldingWeapon("Yoshi");
// Переменная message2 примет значение "Yoshi wakizashi",
// поскольку в вызове функции указаны два аргумента, а
// значение по умолчанию во внимание не принимается
var message2 = getNinjaWieldingWeapon("Yoshi", "wakizashi");
```

Глава 4. Функции для ученика мастера: представление об их вызове

1. Следующая функция вычисляет сумму передаваемых ей аргументов, используя для этой цели объект `arguments`:

```
function sum(){
  var sum = 0;
  for(var i = 0; i < arguments.length; i++){
    sum += arguments[i];
  }

  return sum;
```

```
}

assert(sum(1, 2, 3) === 6, 'Sum of first three numbers is 6');
assert(sum(1, 2, 3, 4) === 10, 'Sum of first four numbers is 10');
```

2. Используя оставшиеся параметры, представленные в предыдущей главе, перепишите функцию `sum()` таким образом, что не пользоваться в ней объектом `arguments`.

3. **Ответ.** В объявление функции следует ввести оставшийся параметр и немного откорректировать ее тело, как выделено ниже полужирным.

```
function sum(... numbers){
  var sum = 0;
  for(var i = 0; i < numbers.length; i++){
    sum += numbers[i];
  }
  return sum;
}

assert(sum(1, 2, 3) === 6, 'Sum of first three numbers is 6');
assert(sum(1, 2, 3, 4) === 10, 'Sum of first four numbers is 10');
```

4. Какие значения примут переменные `ninja` и `samurai` после выполнения приведенного ниже фрагмента кода?

```
function getSamurai(samurai){
  "use strict"

  arguments[0] = "Ishida";

  return samurai;
}

function getNinja(ninja){
  arguments[0] = "Fuma";
  return ninja;
}

var samurai = getSamurai("Toyotomi");
var ninja = getNinja("Yoshi");
```

Ответ. Переменная `samurai` примет значение `"Toyotomi"`, а переменная `ninja` — значение `"Fuma"`. Функция `getSamurai()` выполняется в *строгом* режиме, и поэтому объект `arguments` не присваивает псевдонимы параметрам данной функции. Следовательно, изменение значения ее первого аргумента не приведет к изменению значения параметра `samurai`. А функция `getNinja()` выполняется в *нестрогом* режиме, и поэтому любые изменения в объекте `arguments` отразятся и на ее параметрах.

5. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
function whoAmI1(){
  "use strict";
```

```
  return this;
}

function whoAmI2(){
  return this;
}

assert(whoAmI1() === window, "Window?"); // не пройдет
assert(whoAmI2() === window, "Window?"); // пройдет
```

6. **Ответ.** Функция `whoAmI1()` выполняется в *строгом* режиме. Когда она вызывается как функция, параметр `this` принимает неопределенное значение (`undefined`) и не содержит ссылку на глобальный объект `window`, и поэтому первое утверждение *не*пройдет. Если же данная функция выполняется в *нестрогом* режиме и вызывается как функция, то параметр `this` будет содержать ссылку на объект `window` (при выполнении рассматриваемого здесь фрагмента кода в браузере), и поэтому второе утверждение пройдет.
7. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
var ninja1 = {
  whoAmI: function(){
    return this;
  }
};

var ninja2 = {
  whoAmI: ninja1.whoAmI
};

var identify = ninja2.whoAmI;

// Это утверждение пройдет: функция whoAmI() вызывается
// как метод объекта ninja1
assert(ninja1.whoAmI() === ninja1, "ninja1?");

// А это утверждение не пройдет: функция whoAmI() вызывается
// как метод объекта ninja2
assert(ninja2.whoAmI() === ninja1, " ninja1 again?");

// И это утверждение не пройдет: из переменной identify
// функция whoAmI() вызывается как функция, а поскольку она
// выполняется в нестрогом режиме, то параметр this ссылается
// на глобальный объект window
assert(identify() === ninja1, "ninja1 again?");

// А это утверждение пройдет: метод call() предоставляет контекст
// для функции whoAmI(), а параметр this ссылается на объект ninja2
assert(ninja1.whoAmI.call(ninja2) === ninja2, "ninja2 here?");
```

8. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
function Ninja(){
  this.whoAmI = () => this;
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};

// Это утверждение пройдет: функция whoAmI() является
// стрелочной и наследует свой контекст из того контекста,
// где она была создана. А поскольку она была создана при
// построении объекта ninja1, то параметр this будет всегда
// ссылаться на объект ninja1
assert(ninja1.whoAmI() === ninja1, "ninja1 here?");

// А это утверждение не пройдет: параметр this будет всегда
// ссылаться на объект ninja1
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

9. Какие утверждения пройдут при выполнении приведенного ниже фрагмента кода?

```
function Ninja(){
  this.whoAmI = function(){
    return this;
  }.bind(this);
}

var ninja1 = new Ninja();
var ninja2 = {
  whoAmI: ninja1.whoAmI
};

// Это утверждение пройдет: функция, присваиваемая
// свойству whoAmI, привязана к объекту ninja1, т.е. к значению
// параметра this при вызове конструктора.
// Параметр this будет всегда ссылаться на объект ninja1
assert(ninja1.whoAmI() === ninja1, "ninja1 here?");

// А это утверждение не пройдет: параметр this в функции whoAmI()
// будет всегда ссылаться на объект ninja1, поскольку whoAmI()
// является привязанной функцией
assert(ninja2.whoAmI() === ninja2, "ninja2 here?");
```

Глава 5. Функции для мастера: замыкания и области видимости

1. Замыкания предоставляют функциям доступ к внешним переменным, находившимся в области видимости:

   Ответ. При определении функции (вариант **а**)).

2. Замыкания требуют затрат:

   Ответ. Оперативной памяти (вариант **б**)), поскольку замыкания поддерживают активными переменные, находившиеся в области видимости при определении функции.

3. Отметьте в следующем примере кода идентификаторы, доступные через замыкания.

```
function Samurai(name) {
  var weapon = "katana";

  this.getWeapon = function(){
    // здесь получается доступ к локальной переменной weapon
    return weapon;
  };

  this.getName = function(){
    // здесь получается доступ к параметру функции name
    return name;
  }

  this.message = name + " wielding a " + weapon;

  this.getMessage = function(){
    // по ссылке this.message нельзя получить доступ через
    // замыкание, ведь это свойство объекта, а не переменная
    return this.message;
  }
}

var samurai = new Samurai("Hattori");

samurai.getWeapon();
samurai.getName();
samurai.getMessage();
```

4. Сколько контекстов выполнения создается в приведенном ниже фрагменте кода и каков наибольший размер стека контекстов выполнения?

```
function perform(ninja) {
  sneak(ninja);
  infiltrate(ninja);
}

function sneak(ninja) {
  return ninja + " skulking";
}

function infiltrate(ninja) {
  return ninja + " infiltrating";
}

perform("Kuma");
```

Ответ. Стек контекстов выполнения достигает наибольшего размера, равного 3, в следующих случаях:

- глобальный код -> функция `perform()` -> функция `sneak()`
- глобальный код -> функция `perform()` -> функция `infiltrate()`

5. Какое ключевое слово JavaScript позволяет определить переменные, которым нельзя присвоить снова совершенно новое значение?

   Ответ. Ключевое слово `const` не позволяет присваивать определяемым с его помощью переменным новые значения.

6. В чем отличие ключевых слов `var` и `let`?

   Ответ. Ключевое слово `var` служит для определения переменных только в области видимости функции или глобальной области видимости, тогда как ключевое слово `let` позволяет определить переменные в области видимости блока кода, функции и глобальной области видимости.

7. Где и почему генерируется исключение в приведенном ниже фрагменте кода?

```
getNinja();
getSamurai();

function getNinja() {
  return "Yoshi";
}

var getSamurai = () => "Hattori";
```

   Ответ. Исключение будет сгенерировано при попытке вызвать функцию `getSamurai()`. С одной стороны, функция `getNinja()` определяется путем объявления и будет доступна перед выполнением какого-нибудь кода. Эту функцию можно вызвать прежде, чем в коде будет достигнуто ее объявление. А с другой стороны, стрелочная функция создается в тот момент, когда ее достигает выполнение кода, и поэтому она окажется неопределенной при попытке вызвать ее.

Глава 6. Функции на перспективу: генераторы и обещания

1. Какие значения примут переменные `a1`–`a4` после выполнения приведенного ниже фрагмента кода?

```
function *EvenGenerator(){
  let num = 2;
  while(true){
    yield num;
    num = num + 2;
  }
}
```

```
let generator = EvenGenerator();

// первым возвращается значение 2
let a1 = generator.next().value;

// вторым возвращается значение 4
let a2 = generator.next().value;

// далее возвращается значение 2, поскольку
// запущен новый генератор
let a3 = EvenGenerator().next().value;

// и, наконец, возвращается значение 6 как
// результат возврата к первому генератору
let a4 = generator.next().value;
```

2. Каким окажется содержимое массива `ninjas` после выполнения приведенного ниже фрагмента кода? (*Подсказка:* подумайте, как организовать цикл `for-of` с помощью цикла `while`.)

```
function* NinjaGenerator(){
  yield "Yoshi";
  return "Hattori";
  yield "Hanzo";
}

var ninjas = [];
for(let ninja of NinjaGenerator()){
  ninjas.push(ninja);
}

ninjas;
```

Ответ. Массив `ninjas` будет содержать только значение `"Yoshi"`. Это происходит потому, что обход генератора в цикле `for-of` осуществляется до тех пор, пока он не завершит свое выполнение, ничего не возвратив. И это произойдет либо тогда, когда в генераторе больше не останется кода для выполнения, либо тогда, когда в нем встретится оператор `return`.

3. Какие значения примут переменные `a1-a2` после выполнения приведенного ниже фрагмента кода?

```
function *Gen(val){
  val = yield val * 2;
  yield val;
}

let generator = Gen(2);
// Значение 4. Первое значение свойства value передается
// через метод next(), а значение 3 игнорируется, поскольку
// генератор еще не начал свое выполнение и отсутствует
// ожидающее выражение yield. Генератор создается при
// значении val = 2, и поэтому первым получается значение
```

```
// для выражения val * 2, т.е. 2*2 == 4
let a1 = generator.next(3).value;
// Значение 5. Передача значения 5 в качестве аргумента
// методу next() означает, что ожидающее выражение yield
// получит значение 5:
// yield val * 2 == 5.
// А поскольку это значение присваивается далее переменной val,
// то из следующего выражения yield:
// yield val;
// возвратится значение 5
let a2 = generator.next(5).value;
```

4. Каким окажется результат выполнения приведенного ниже фрагмента кода?

```
const promise = new Promise((resolve, reject) => {
  reject("Hattori"); // обещание явно отклонено
});

// будет вызван обработчик событий
promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

5. Каким окажется результат выполнения приведенного ниже фрагмента кода?

```
const promise = new Promise((resolve, reject) => {
  // обещание было разрешено явным образом
  resolve("Hattori");
  // как только обещание будет разрешено, его нельзя
  // изменить, а его отклонение через 500 мс не возымеет
  // никакого действия
  setTimeout(()=> reject("Yoshi"), 500);
});

// будет вызван обработчик успешно разрешенного обещания
promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

Глава 7. Объектная ориентация с помощью прототипов

1. Какое из приведенных ниже свойств указывает на объект, поиск в котором будет осуществлен, если у целевого объекта отсутствует искомое свойство?

   Ответ. `prototype` (вариант **в**)).

2. Какое значение примет переменная `a1` после выполнения приведенного ниже фрагмента кода?

```
function Ninja(){}
  Ninja.prototype.talk = function (){
```

```
    return "Hello";
};

const ninja = new Ninja();
const a1 = ninja.talk(); // значение "Hello"
```

Ответ. Переменная `a1` примет строковое значение `"Hello"`. Несмотря на то что объект `ninja` не обладает методом `talk()`, им обладает его прототип.

3. Какое значение примет переменная `a1` после выполнения приведенного ниже фрагмента кода?

```
function Ninja(){}
Ninja.message = "Hello";

const ninja = new Ninja();

const a1 = ninja.message;
```

Ответ. Значение переменной `a1` окажется неопределенным (`undefined`). Свойство `message` определяется в функции-конструкторе объектов типа `Ninja` и недоступно через объект `ninja`.

4. Поясните отличия в методе `getFullName()`, обнаруживаемые в двух приведенных ниже фрагментах кода.

```
// Первый фрагмент
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;

  this.getFullName = function () {
    return this.firstName + " " + this.lastName;
  }
}

// Второй фрагмент
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.getFullName = function () {
  return this.firstName + " " + this.lastName;
}
```

Ответ. В первом фрагменте данного кода метод `getFullName()` определяется непосредственно для экземпляра, создаваемого конструктором объектов типа `Person`. Каждый экземпляр, создаваемый конструктором объектов типа `Person`, получает свой собственный метод `getFullName()`. А во втором фрагменте данного кода метод `getFullName()` определяется для прототипа функции-конструктора объектов типа `Person`. Этот единственный метод будет доступен всем экземплярам, создаваемым функцией-конструктором объектов типа `Person`.

5. На что будет указывать свойство `ninja.constructor` после выполнения приведенного ниже фрагмента кода?

```
function Person() { }
function Ninja() { }
const ninja = new Ninja();
```

Ответ. При доступе к свойству `ninja.constructor` оно обнаруживается в прототипе объекта `ninja`. А поскольку объект `ninja` был создан функцией-конструктором объектов типа `Ninja`, то свойство `ninja.constructor` указывает на конструктор объектов типа `Ninja`.

6. На что будет указывать свойство `ninja.constructor` после выполнения приведенного ниже фрагмента кода?

```
function Person() { }
function Ninja() { }
Ninja.prototype = new Person();
const ninja = new Ninja();
```

Ответ. Свойство `constructor` относится к объекту-прототипу, созданному функцией-конструктором. В данном примере кода встроенный прототип функции-конструктора объектов типа `Ninja` переопределяется новым объектом типа `Person`. Следовательно, когда объект `ninja` создается конструктором объектов типа `Ninja`, в его прототипе устанавливается новый объект `person`. И, наконец, при доступе к свойству `constructor` объекта `ninja` фактически происходит обращение к его прототипу (т.е. к новому объекту `person`), поскольку у самого объекта `ninja` отсутствует собственное свойство `constructor`. Но и у объекта `person` отсутствует свойство `constructor`, поэтому происходит обращение к его прототипу, (т.е. к объекту `Person.prototype`). И у этого объекта имеется свойство `constructor`, ссылающееся на функцию-конструктор объектов типа `Person`. Данный пример наглядно показывает, почему следует очень аккуратно обращаться со свойством `constructor`. Несмотря на то что объект `ninja` был создан функцией-конструктором объектов типа `Ninja`, доступное по умолчанию свойство `Ninja.prototype` указывает на функцию-конструктор объектов типа `Person` из-за особенностей переопределения.

7. Поясните, каким образом операция `instanceof` действует в следующем примере кода:

```
function Warrior() { }

function Samurai() { }
Samurai.prototype = new Warrior();

var samurai = new Samurai();

samurai instanceof Warrior; // пояснить
```

Ответ. В данной операции `instanceof` проверяется, находится ли прототип функции, указываемый в правой ее части, в цепочке прототипов объекта, указываемого в левой ее части. Объект, указываемый в левой части рассматриваемой здесь операции `instanceof`, создается функцией-конструктором объектов типа `Samurai`, а у его прототипа имеется новый объект `warrior`, прототипом которого служит прототип функции-конструктора объектов типа `Warrior` (`Warrior.prototype`). А в правой части данной операции указывается функция-конструктор объектов типа `Warrior`. Таким образом, в данном примере кода из операции `instanceof` возвратится логическое значение `true`, поскольку прототип функции в правой ее части (`Warrior.prototype`) может быть обнаружен в цепочке прототипов объекта, указанного в левой ее части.

8. Преобразуйте следующий фрагмент кода из стандарта ES6 в стандарт ES5:

```
class Warrior {
  constructor(weapon){
    this.weapon = weapon;
  }

  wield() {
    return "Wielding " + this.weapon;
  }

  static duel(warrior1, warrior2){
    return warrior1.wield() + " " + warrior2.wield();
  }
}
```

Ответ. Указанный выше фрагмент кода можно преобразовать в стандарт ES5 следующим образом:

```
function Warrior(weapon) {
  this.weapon = weapon;
}

Warrior.prototype.wield = function () {
  return "Wielding " + this.weapon;
};

Warrior.duel = function (warrior1, warrior2) {
  return warrior1.wield() + " " + warrior2.wield();
};
```

Глава 8. Управление доступом к объектам

1. В каком из выражений в двух последних строках приведенного ниже фрагмента кода могут быть сгенерированы исключения при его выполнении и почему это может произойти?

```
const ninja = {
  get name() {
    return "Akiyama";
  }
}

ninja.name();
const name = ninja.name;
```

Ответ. В результате вызова метода `ninja.name()` генерируется исключение, поскольку у объекта `ninja` отсутствует метод `name()`. А при доступе к свойству `ninja.name` в выражении `const name = ninja.name` благополучно активизируется метод получения и переменной `name` присваивается значение `"Akiyama"`.

2. Какой механизм позволяет методу получения обратиться к закрытой объектной переменной в приведенном ниже фрагменте кода?

```
function Samurai() {
  const _weapon = "katana";
  Object.defineProperty(this, "weapon", {
    get: () => _weapon
  });
}
const samurai = new Samurai();
assert(samurai.weapon === "katana", "A katana wielding samurai");
```

Ответ. Замыкания предоставляют методам получения доступ к закрытым объектным переменным. В данном случае вокруг закрытой переменной `_weapon`, определяемой в функции-конструкторе, в методе `get()` образуется замыкание, поддерживающее активной переменную `_weapon`.

3. Какое из приведенных ниже утверждений пройдет?

```
const daimyo = { name: "Matsu", clan: "Takasu"};
const proxy = new Proxy(daimyo, {
  get: (target, key) => {
    if(key === "clan"){
      return "Tokugawa";
    }
  }
});

assert(daimyo.clan === "Takasu",
      "Matsu of clan Takasu");    // пройдет
assert(proxy.clan === "Tokugawa",
      "Matsu of clan Tokugawa?"); // пройдет

proxy.clan = "Tokugawa";

assert(daimyo.clan === "Takasu",
       "Matsu of clan Takasu");    // не пройдет
assert(proxy.clan === "Tokugawa",
       "Matsu of clan Tokugawa?"); // пройдет
```

Ответ. Первое утверждение пройдет, потому что у объекта `daimyo` имеется свойство `clan` со значением `"Takasu"`. И второе утверждение пройдет, потому что свойство `clan` доступно через прокси-объект с методом перехвата `get()`, который всегда возвращает `"Tokugawa"` в качестве значения свойства `clan`.

4. Когда вычисляется выражение `proxy.clan = "Tokugawa"`, значение `"Tokugawa"` сохраняется в свойстве `clan` объекта `daimyo`, поскольку у прокси-объекта отсутствует метод перехвата `set()`, а следовательно, стандартное действие по установке значения в данном свойстве выполняется над целевым объектом `daimyo`.
5. Третье утверждение не пройдет, поскольку в свойстве `clan` объекта `daimyo` хранится значение `"Tokugawa"`, а не значение `"Takasu"`.
6. И, наконец, четвертое утверждение пройдет, потому что прокси-объект всегда возвращает значение `"Tokugawa"`, независимо от того, какое именно значение хранится в свойстве `clan` целевого объекта.
7. Какое из приведенных ниже утверждений пройдет?

```
const daimyo = { name: "Matsu", clan: "Takasu", armySize: 10000};
const proxy = new Proxy(daimyo, {
  set: (target, key, value) => {
    if(key === "armySize") {
      const number = Number.parseInt(value);
      if(!Number.isNaN(number)){
        target[key] = number;
      }
    } else {
      target[key] = value;
    }
  },
});

assert(daimyo.armySize === 10000,
       "Matsu has 10 000 men at arms"); // пройдет
assert(proxy.armySize === 10000,
      "Matsu has 10 000 men at arms");  // пройдет

proxy.armySize = "large";
assert(daimyo.armySize === "large",
       "Matsu has a large army");  // не пройдет

daimyo.armySize = "large";
assert(daimyo.armySize === "large",
       "Matsu has a large army");  // пройдет
```

Ответ. Первое утверждение пройдет, а свойство `armySize` объекта `daimyo` принимает значение **`10000`**. И второе утверждение пройдет, поскольку в прокси-объекте не определен метод перехвата `get()`, и в связи с этим возвращается значение свойства `armySize` целевого объекта

daimyo. Когда вычисляется выражение proxy.armySize = "large";, в прокси-объекте активизируется метод перехвата set(). В этом методе проверяется, является ли переданное ему значение числовым, и только в том случае, если это значение действительно оказывается числовым, оно присваивается свойству целевого объекта. В данном случае значение, переданное методу перехвата set(), не является числовым, и поэтому никаких изменений в свойстве armySize целевого объекта daimyo не происходит. Именно по этой причине не пройдет третье утверждение, где предполагается изменение в данном свойстве. Далее в выражении daimyo.armySize = "large"; выполняется непосредственная запись значения свойства armySize в обход прокси-объекта. Следовательно, последнее утверждение пройдет.

Глава 9. Работа с коллекциями

1. Каким окажется содержимое массива samurai после выполнения следующего фрагмента кода?

```
const samurai = ["Oda", "Tomoe"];
samurai[3] = "Hattori";
```

Ответ. Содержимое массива samurai окажется следующим: ["Oda", "Tomoe", undefined, "Hattori"]. В начале данного массива устанавливаются значения "Oda" и "Tomoe" по индексам **0** и **1** соответственно. А затем в массив samurai вводится новое имя самурая Hattori по индексу **3**, которым расширяется данный массив. В то же время значение по индексу **2** остается неопределенным (undefined).

2. Каким окажется содержимое массива ninjas после выполнения следующего фрагмента кода?

```
const ninjas = [];

ninjas.push("Yoshi");

ninjas.unshift("Hattori");

ninjas.length = 3;

ninjas.pop();
```

Ответ. Содержимое массива ninjas будет следующим: ["Hattori", "Yoshi"]. Сначала в данном примере кода создается пустой массив ninjas, затем в его конец добавляется значение "Yoshi" методом push(), а в его начало — значение "Hattori" методом unshift(). Далее установкой значения **3** в свойстве length данный массив расширяется, и по индексу **2** будет храниться неопределенное значение (undefined). А в результате вызова метода pop() неопределенное значение удаляется из массива ninjas, где в конечном итоге остается следующее содержимое: ["Hattori", "Yoshi"].

3. Каким окажется содержимое массива `samurai` после выполнения следующего фрагмента кода?

```
const samurai = [];

samurai.push("Oda");
samurai.unshift("Tomoe");
samurai.splice(1, 0, "Hattori", "Takeda");
samurai.pop();
```

Ответ. Содержимое массива `samurai` будет следующим: `["Tomoe", "Hattori", "Takeda"]`. Сначала в данном примере кода создается пустой массив `samurai`, затем в его конец добавляется значение `"Oda"` методом `push()`, а в его начало — значение `"Tomoe"` методом `unshift()`. Метод `splice()` не удаляет ничего из массива, а просто помещает значения `"Hattori"` и `"Takeda"` по индексу 1 (после `"Tomoe"`). Вызов последнего метода `pop()` удаляет значение `"Oda"` из конца массива.

4. Что останется в переменных `first`, `second` и `third` после выполнения следующего фрагмента кода?

```
const ninjas = [{name:"Yoshi", age: 18},
                {name:"Hattori", age: 19},
                {name:"Yagyu", age: 20}];

const first = ninjas.map(ninja => ninja.age);
const second = first.filter(age => age % 2 == 0);
const third = first.reduce((aggregate, item) => aggregate + item, 0);
```

Ответ. Переменная `first`: `[18, 19, 20]`; переменная `second`: `[18, 20]`; переменная `third`: **57**.

5. Что останется в переменных `first` и `second` после выполнения следующего фрагмента кода?

```
const ninjas = [{ name: "Yoshi", age: 18 },
                { name: "Hattori", age: 19 },
                { name: "Yagyu", age: 20 }];

const first = ninjas.some(ninja => ninja.age % 2 == 0);
const second = ninjas.every(ninja => ninja.age % 2 == 0);
```

Ответ. Переменная `first`: логическое значение `true`; переменная `second`: логическое значение `false`.

6. Какие из приведенных ниже утверждений пройдут?

```
const samuraiClanMap = new Map();

const samurai1 = { name: "Toyotomi"};
const samurai2 = { name: "Takeda"};
const samurai3 = { name: "Akiyama"};

const oda = { clan: "Oda"};
const tokugawa = { clan: "Tokugawa"};
const takeda ={clan: "Takeda"};
```

```
samuraiClanMap.set(samurai1, oda);
samuraiClanMap.set(samurai2, tokugawa);
samuraiClanMap.set(samurai2, takeda);

assert(samuraiClanMap.size === 3,
       "There are three mappings");
assert(samuraiClanMap.has(samurai1),
       "The first samurai has a mapping");
assert(samuraiClanMap.has(samurai3),
       "The third samurai has a mapping");
```

Ответ. Первое утверждение не пройдет, поскольку отображение для переменной `samurai2` создано дважды. А второе утверждение пройдет, поскольку добавлено отображение для переменной `samurai1`. И третье утверждение не пройдет, поскольку для переменной `samurai3` вообще не создано отображение.

7. Какие из приведенных ниже утверждений пройдут?

```
const samurai = new Set("Toyotomi", "Takeda", "Akiyama", "Akiyama");

assert(samurai.size === 4, "There are four samurai in the set");

samurai.add("Akiyama");
assert(samurai.size === 5, "There are five samurai in the set");

assert(samurai.has("Toyotomi", "Toyotomi is in!");
assert(samurai.has("Hattori", "Hattori is in!");
```

Ответ. Первое утверждение не пройдет, поскольку значение `"Akiyama"` вводится в множество только один раз. И второе утверждение не пройдет, поскольку попытка еще раз ввести значение `"Akiyama"` в множество не изменит ни его содержимое, ни его длину. Предпоследнее утверждение пройдет, поскольку значение `"Toyotomi"` входит в наше множество. Последнее утверждение не проходит, потому что значение `"Hattori"` не принадлежит множеству.

Глава 10. Овладение регулярными выражениями

1. Какими из приведенных ниже языковых средств могут быть составлены регулярные выражения в JavaScript?

   а) Литералы регулярных выражений.

   б) Встроенный конструктор объектов типа `RegExp`.

   в) Встроенный конструктор объектов типа `RegularExpression`.

   Ответ. С помощью литералов регулярных выражений (вариант **а**)) и встроенного конструктора объектов типа `RegExp` (вариант **б**)). А вариант **в**) ответа неверный, поскольку встроенного конструктора объектов типа `RegularExpression` не существует.

2. Что из приведенного ниже является литералом регулярного выражения?

а) `/test/`

б) `\test\`

в) `new RegExp("test");`

Ответ. В языке JavaScript литерал регулярного выражения ограничивается двумя знаками косой черты (вариант **а**)).

3. Выберите из приведенных ниже флагов те, которые правильно указаны в регулярном выражении.

а) `test/g`

б) `g/test/`

в) `new RegExp("test", "gi");`

Ответ. В литералах регулярных выражений флаги указываются после закрывающей косой черты: `test/g` (вариант **а**)). А в конструкторах объектов типа `RegExp` они передаются в качестве второго аргумента: `new RegExp("test", "gi");` (вариант **в**)).

4. С какой из приведенных ниже символьных строк совпадает регулярное выражение `/def/`?

а) Одна из символьных строк `"d"`, `"e"`, `"f"`

б) `"def"`

в) `"de"`

Ответ. Регулярное выражение `/def/` совпадает только с символьной строкой `"def"`, где подряд следуют символы `d`, `e` и `f` (вариант **б**)).

5. С какой из приведенных ниже символьных строк совпадает регулярное выражение `/[^abc]/`?

а) Одна из символьных строк `"a"`, `"b"`, `"c"`

б) Одна из символьных строк `"d"`, `"e"`, `"f"`

в) Символьная строка `"ab"`

Ответ. Регулярное выражение `/[^abc]/` совпадает с одной из символьных строк `"d"`, `"e"`, `"f"`, состоящей из единственного символа, не являющегося символом `a`, `b` или `c` (вариант **б**)).

6. С каким из приведенных ниже регулярных выражений совпадает символьная строка `"hello"`?

а) `/hello/`

б) `/hell?o/`

в) `/hel*o/`

г) `/[hello]/`

Ответ. С регулярным выражением `/hello/` точно совпадает только строка `"hello"` (вариант **а**)). С регулярным выражением `/hell?o/` совпадает строка `"hello"` или `"helo"`, а второй символ `l` необязателен (ва-

риант **б**)). А с регулярным выражением `/hel*o/` после первого символа `l` совпадает любое количество следующих далее символов `l` (вариант **в**)).

7. С какой из приведенных ниже символьных строк совпадает регулярное выражение `/(cd)+(de)*/`?

а) `"cd"`

б) `"de"`

в) `"cdde"`

г) `"cdcd"`

д) `"ce"`

е) `"cdcddedede"`

Ответ. Регулярное выражение `/(cd)+(de)*/` совпадает с одним или несколькими вхождениями символов `cd`, после которых следует любое количество вхождений символов `de` (варианты **а**), **в**), **г**) и **е**)).

8. Каким из приведенных ниже знаков можно обозначить альтернативные варианты в регулярном выражении?

а) **#**

б) **&**

в) **|**

Ответ. Альтернативные варианты в регулярном выражении можно обозначить знаком вертикальной черты (**|**; вариант **в**)).

9. Каким из приведенных ниже способов можно сослаться на первую совпавшую цифру в регулярном выражении `/([0-9])2/`?

а) `/0`

б) `/1`

в) `\0`

г) `\1`

Ответ. `\1` (вариант **г**)).

10. С каким из приведенных ниже чисел совпадет регулярное выражение `/([0-5])6\1/`?

а) **060**

б) **16**

в) **261**

г) **565**

Ответ. Первым символом в регулярном выражении `/([0-5])6\1/` является цифра от **0** до **5**, вторым символом — цифра **6**, а третьим символом — первая совпавшая цифра, поэтому данное регулярное выражение совпадет с числами **060** и **565** (варианты **а**) и **г**)).

11. С какой из приведенных ниже символьных строк совпадет регулярное выражение `/(?:ninja)-(trick)?-\1/?`

а) `"ninja-"`

б) `"ninja-trick-ninja"`

в) `"ninja-trick-trick"`

Ответ. Первая группа `(?:ninja)` в регулярном выражении `/(?:ninja)-(trick)?-\1/` является нефиксируемой, тогда как вторая группа `(trick)?` — фиксируемой и необязательной. Но если обнаруживается вторая группа, то в конечном итоге получается обратная ссылка на нее. Следовательно, данное регулярное выражение совпадет с символьными строками `"ninja-"` и `"ninja-trick-trick"` (варианты **а)** и **в)**).

12. К какому из приведенных ниже результатов приведет вызов `"012675".replace(/[0-5]/g, "a")`?

а) `"aaa67a"`

б) `"a12675"`

в) `"a1267a"`

Ответ. В данном вызове все вхождения цифр от `0` до `5` заменяются буквой **a**, поэтому в результате получится символьная строка `"aaa67a"` (вариант **а)**).

Глава 11. Методики модуляризации кода

1. Какой из перечисленных ниже механизмов допускает наличие закрытых переменных модуля в проектном шаблоне `Модуль`?

а) Прототипы

б) Замыкания

в) Обещания

Ответ. В проектном шаблоне `Модуль` замыкания позволяют скрывать внутренние элементы модуля, которые поддерживаются активными с помощью методов из интерфейса API данного модуля (вариант **б)**).

2. В приведенном ниже фрагменте кода применяются модули, внедренные в стандарт ES6. Какие из перечисленных ниже идентификаторов могут быть доступны, если импортировать модуль?

```
const spy = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
export const general = "Minamoto";
```

а) `spy`

б) `command`

в) `general`

Ответ. За пределами модуля может быть доступным только идентификатор `general`, поскольку это единственный экспортированный идентификатор (вариант **с**)).

3. В приведенном ниже фрагменте кода применяются модули, внедренные в стандарт ES6. Какие из перечисленных ниже идентификаторов могут быть доступны, если импортировать модуль?

```
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general = "Minamoto";

export {ninja as spy};
```

а) `spy`

б) `command`

в) `general`

г) `ninja`

4. **Ответ.** За пределами модуля может быть доступным только идентификатор `spy`, поскольку это единственный идентификатор, экспортированный в качестве псевдонима переменной `ninja` (вариант **а**)).

5. Какой из перечисленных ниже видов импорта модулей допустим?

```
// Файл модуля: personnel.js
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general = "Minamoto";

export {ninja as spy};
```

а) `import {ninja, spy, general} from "personnel.js"`

б) `import * as Personnel from "personnel.js"`

в) `import {spy} from "personnel.js"`

Ответ. Первый вид импорта (вариант **а**)) недопустим, поскольку в модуле `personnel` не экспортируются идентификаторы `ninja` и `general`. Второй вид импорта (вариант **б**)) допустим, поскольку импортируется весь модуль, доступный через объект типа `Personnel`. И третий вид импорта (вариант **в**)) допустим, поскольку импортируется экспортированный идентификатор `spy`.

6. Какой из перечисленных ниже операторов импортирует класс `Ninja` в приведенном ниже фрагменте кода?

```
// Ninja.js
export default class Ninja {
```

```
  skulk(){ return "skulking"; }
}
```

а) `import Ninja from "Ninja.js"`

б) `import * as Ninja from "Ninja.js"`

в) `import * from "Ninja.js"`

Ответ. Первый вид импорта (вариант **а**)) допустим, поскольку импортируется то, что экспортируется по умолчанию. И второй вид импорта (вариант **б**)) допустим, поскольку экспортируется весь модуль. А третий вид импорта (вариант **в**)) недопустим, поскольку он составлен синтаксически неверно (после знака ***** должно следовать выражение `as` *`ИмяКласса`*).

Глава 12. Работа с моделью DOM

1. Какие утверждения пройдут в приведенном ниже фрагменте кода?

```
<div id="samurai"></div>
<script>
  const element = document.querySelector("#samurai");

  assert(element.id === "samurai", "property id is samurai");
  assert(element.getAttribute("id") === "samurai",
         "attribute id is samurai");

  element.id = "newSamurai";

  assert(element.id === "newSamurai", "property id is newSamurai");
  assert(element.getAttribute("id") === "newSamurai",
         "attribute id is newSamurai");
</script>
```

Ответ. В данном фрагменте кода проходят все утверждения. Атрибут `id` и свойство `id` связаны вместе, поэтому изменение в одном из них отражается на другом.

2. Каким из перечисленных ниже способов можно получить доступ к свойству `border-width` стилевого оформления элемента разметки в приведенном ниже фрагменте кода?

```
<div id="element" style="border-width: 1px;
                         border-style:solid; border-color: red">
</div>
<script>
  const element = document.querySelector("#element");
</script>
```

а) `element.border-width`

б) `element.getAttribute("border-width");`

в) `element.style["border-width"];`

г) `element.style.borderWidth;`

Ответ. Выражение `element.border-width` не имеет особого смысла, поскольку в нем вычисляется разность значений свойства `element.border` и переменной `width`, а это совершенно не то, что требуется в данном случае. В следующем далее выражении `element.getAttribute("border-width");` извлекается атрибут элемента HTML-разметки, а не свойство стилевого оформления. И, наконец, в двух последних выражениях (варианты **в)** и **г)**) получается значение **`1px`** свойства стилевого оформления.

3. Какой из перечисленных ниже встроенных методов позволяет получить все стили, применяемые к определенному элементу (т.е. стили, предоставляемые браузером; стили, применяемые через таблицы стилей; а также свойства, устанавливаемые через атрибут стилевого оформления)?

   а) `getStyle()`

   б) `getAllStyles()`

   в) `getComputedStyle()`

   Ответ. Для получения вычисленного стиля оформления определенного элемента HTML-разметки может быть использован только встроенный метод `getComputedStyle()` (вариант **в)**). А остальные методы не входят в состав стандартного интерфейса API.

4. Когда происходит перегрузка верстки?

   Ответ. Перегрузка верстки происходит при выполнении в прикладном коде последовательного ряда операций чтения и записи данных в модели DOM, которые всякий раз вынуждают браузер выполнять перерасчет данных верстки. И это приводит к замедлению реакции веб-приложений на действия пользователей.

Глава 13. Особенности обработки событий

1. Почему так важно, чтобы постановка задач в очередь происходила за пределами цикла ожидания событий?

   Ответ. Если бы процесс постановки задач в очередь происходил в цикле ожидания событий, то любые события, наступившие во время выполнения кода на JavaScript, были бы проигнорированы, а это никуда не годится.

2. Почему так важно, чтобы каждый шаг цикла ожидания событий занимал не больше 16,7 мс?

3. **Ответ.** Чтобы добиться плавного воспроизведения анимации, браузер пытается пересчитывать и отображать содержимое веб-страницы 60 раз в секунду. Пересчет элементов интерфейса веб-страницы выполняется в конце цикла ожидания событий, и поэтому каждый шаг этого цикла должен длиться не больше 16,7 мс, чтобы приложения работали без задержек, оперативно реагируя на действия пользователей.

4. Какой из перечисленных ниже результатов будет выведен на консоль через 2 секунды после начала выполнения приложения?

```
setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);
```

а) `Timeout Interval Interval Interval Interval`

б) `Interval Timeout Interval Interval Interval`

в) `Interval Timeout Timeout`

Ответ. `Interval Timeout Interval Interval Interval` (вариант **б**)). Как известно, метод `setInterval()` инициирует вызов обработчика события на постоянной основе через заданный интервал времени вплоть до его отмены. В то же время, метод `setTimeout()` инициирует лишь однократный вызов обработчика события по истечении указанного времени задержки. В нашем примере сначала через 500 мс будет вызвана функция обратного вызова, установленная методом `setInterval()`. Затем, через 1000 мс делается обратный вызов, установленный методом `setTimeout()` и сразу после этого еще один обратный вызов, установленный методом `setInterval()`. А далее делаются еще два обратных вызова, установленных в методе `setInterval()` через 1500 и 2000 мс соответственно.

5. Какой из перечисленных ниже результатов будет выведен на консоль через 2 секунды после начала выполнения приложения?

```
const timeoutId = setTimeout(function(){
  console.log("Timeout ");
}, 1000);

setInterval(function(){
  console.log("Interval ");
}, 500);

clearTimeout(timeoutId);
```

- а) `Interval Timeout Interval Interval Interval`
- б) `Interval`
- в) `Interval Interval Interval Interval`

Ответ. `Interval Interval Interval Interval` (вариант **в**)). Таймер, запущенный в результате вызова метода `setTimeout()`, будет сброшен до того, как истечет указанное время задержки (практически сразу после вызова этого метода). Поэтому в данном случае делаются лишь четыре обратных вызова, установленные в методе `setInterval()`.

6. Какой из перечисленных ниже результатов будет выведен на консоль в результате выполнения приведенного ниже кода и щелчка кнопкой мыши на элементе с идентификатором inner?

```
<body>
  <div id="outer">
    <div id="inner"></div>
  </div>
  <script>
    const innerElement = document.querySelector("#inner");
    const outerElement = document.querySelector("#outer");
    const bodyElement = document.querySelector("body");

    innerElement.addEventListener("click", function(){
      console.log("Inner");
    });

    outerElement.addEventListener("click", function(){
      console.log("Outer");
    }, true);

    bodyElement.addEventListener("click", function(){
      console.log("Body");
    })
  </script>
</body>
```

а) `Inner Outer Body`

б) `Body Outer Inner`

в) `Outer Inner Body`

Ответ. `Outer Inner Body` (вариант **в**)). Обработчики событий от щелчков кнопкой мыши на элементах `innerElement` и `bodyElement` регистрируются в режиме всплывания событий, тогда как обработчик событий от щелчков кнопкой мыши на элементе `outerElement` — в режиме перехвата событий. При обработке события оно распространяется сверху вниз, а все обработчики событий вызываются в режиме перехвата. Первым выводится сообщение `"Outer"`. Как только будет достигнут инициатор события (в данном случае — элемент с идентификатором `inner`), происходит всплывание события снизу вверх. Поэтому вторым выводится сообщение `"Inner"`, а третьим — сообщение `"Body"`.

Глава 14. Стратегии разработки кросс-браузерного кода

1. Что следует принять во внимание, выбирая браузеры для поддержки разрабатываемого веб-приложения?

Ответ. Выбирая конкретные браузеры для поддержки разработки веб-приложений, следует, по меньшей мере, иметь в виду следующее.

- Ожидания и потребности целевой аудитории.
- Доля браузера на рынке.
- Усилия, которые необходимо приложить для поддержки браузера.

2. Объясните осложнения, к которым приводит использование поглощающих идентификаторов.

Ответ. В браузерах ссылки на все элементы разметки формы типа `<input>` сохраняются в виде свойств элемента `<form>`, чтобы к ним можно было легко обращаться. К сожалению, это может привести к переопределению некоторых встроенных свойств вроде `action` или `submit`.

3. Что означает обнаружение функциональных средств?

Ответ. Обнаружение функциональных средств означает возможность выяснить, существуют ли определенные объекты, и если они действительно существуют, то предположить, что они способны предоставить требуемые функциональные возможности. Вместо того чтобы сначала проверить, применяет ли пользователь конкретный браузер, а затем реализовать обходные приемы на основании этой информации, в данном случае проверяется, действует ли определенное функциональное средство именно так, как и предполагалось.

4. Что такое кросс-браузерные полифиллы?

Ответ. Если требуется воспользоваться определенными функциональными возможностями, которые не поддерживаются всеми целевыми браузерами, то можно прибегнуть к методу обнаружения функциональных средств. А если текущий браузер не поддерживает некоторые функциональные возможности, то можно предоставить свою реализацию этих возможностей. И это называется *полифиллом*.

Предметный указатель

A

Asynchronous Module Definition 353

C

Chakra 48
Chrome DevTools 36
CommonJS 353

D

Document object model. См. DOM

F

F12 Developer Tools 36
Firebug 36, 484
Firefox Developer Tools 36

S

Spidermonkey 48

V

V8 48

W

WebKit Inspector 36

А

Анализ производительности
  интерфейсов API 471
  с помощью прокси-объектов 272
  способы измерения 37

Б

Браузеры 27
  инструментальные средства веб-разработки 36
  интерфейсы API 49
  источники сведений о новых выпусках 464
  классификация и матрица поддержки 452
  назначение 33
  несовместимость, вопросы разрешения 35, 452
  области непроверяемых ошибок 470, 471
  обработка событий, порядок 53
  составляющие инфраструктуры 34
  устранение программных ошибок 456
  факторы выбора поддержки 453
Бусидо 24

В

Веб-приложения
  выполнение кода JavaScript 48
  жизненный цикл
    обработка событий 52
    построение страницы, этапы 45
    стадии 43
  построение модели DOM 46
  с ГПИ
    клиентские 43
    пример 43
  средства разработки и отладки 484, 488

Г

Генераторы 32
  как функции, определение 174
  контекст выполнения, отслеживание 187
  обмен данными 183
  определение 171
  перебор возвращаемых результатов 178
  применение, примеры 174, 179, 183
  синтаксис определения 175
  состояния, виды 187
  сочетание с обещаниями 207
  управление с помощью итераторов 176

Глобальный код, определение 139
Глобальный объект 49

Д

Деструктурирование
  массивов 479
  назначение 479
  объектов 479

З

Замыкания 129, 159
  внутренний механизм действия 159
  назначение 130
  применение, примеры 134, 165
  принцип действия 132

И

Идентификаторы
  разрешение 142
  регистрация, процесс 155
  уникальные, генерирование 180
  функций, переопределение 158
Исключения
  генерирование
    для генераторов 185
    при неявном отклонении обещаний 202
Итераторы 176
  назначение 176
  порядок обхода 177
  принцип действия 176

К

Классы
  как синтаксическое удобство 245
  определение методов получения и установки 258
  реализация наследования 248
  создание
    порядок 244
    экземпляров, порядок 244
Ключевые слова
  as, назначение 370
  class, назначение 243
  default, назначение 368
  export, назначение 366
  extends 248
  extends, назначение 249
  function, назначение 77
  get и set, назначение 257
  import, назначение 366
  let и const, применение 152
  new, назначение 135
  static, назначение 246
  super, назначение 248
  this, назначение 436
  var, применение 150
  yield, применение 175
Код функции, определение 139
Конструкторы 112
  объектов типа
    Array, назначение 285
    Map, назначение 309
    Proxy, назначение 266
    RegExp, назначение 326
    Set, назначение 315
  функций 107
    назначение 107
Контекст
  выполнения
    генераторов, применение 187
    глобальный, определение 139
    отслеживание 139
    разрешение идентификаторов 142
    создание 140
    функции, определение 139
  функции
    задание, исходя из способа вызова 113
    назначение 95
    представление через параметр this 95
    привязка методом bind() 124
    принудительная установка 117
    разрешение затруднений 120
    установка явным образом 115

Л

Лексические среды
  вложение кода, типы 143
  внешние, отслеживание 144
  как области видимости 143
  назначение 142
Литералы
  массивов, назначение 285
  объектов
    назначение 123
    усовершенствованные, назначение 481
  регулярных выражений, назначение 326
  функциональные, назначение 75
  шаблонные
    многострочные 478
    назначение 477

М

Массивы
  агрегирование элементов, метод 302
  добавление и удаление элементов, методы 288
  как объекты, особенность 284
  методы
    splice() 290
  назначение 284
  наиболее употребительные операции 291
  отображение, метод 294
  отрицательные индексы, имитация 276
  перебор элементов 292
  поиск элементов, методы 297
  проверка элементов, методы 295
  свойство length, назначение 285
  создание, способы 284
  сортировка, метод 300
  массива
Методы
  addEventListener() 56
  apply(), применение 115
  bind() 124
  call(), применение 115
  catch(), применение 201
  fail() и pass(), применение 149
  forEach(), применение 292
    indexOf() 300
    lastIndexOf() 300
  next(), применение 184
  Object.defineProperty(), применение 237, 261
  Object.setPrototypeOf(), применение 220
  replace(), применение 341
  then(), применение 194
  throw(), применение 185
  доступа, определение 135
  манипулирования таймерами, назначение 424
  массива
    filter() 298
    find() 298
    findIndex() 300
    reduce() 302
  обработки массивов
    повторное использование 303
    применение 287
    разновидности 287
  оперирования множествами, разновидности 316
  отображения, разновидности 311
  перехвата
    назначение 267
    разновидности 269
  получения
    неявный вызов 258
    определение 135, 237
    применение 255, 264
    способы определения 256, 260
  прототипные, применение 245
  статические, применение 245
  установки
    неявный вызов 258
    определение 237

применение 255, 262, 264
способы определения 256, 260
Множества
имитация с помощью объектов 315
методы обработки, разновидности 316
объединение 317
определение 314
пересечение 318
разности 319
свойство size, назначение 316
создание 315
Модель DOM
атрибуты
и свойства элементов 386
специальные, обращение 388
стилевого оформления, обращение 388
видоизменение в коде JavaScript 52, 378, 403
виртуальная, из библиотеки React 406
вставка HTML-кода, способы 379
затирание свойств элементов 462
обход узлов 181
построение 46
преобразование из формата HTML в структуру DOM, процесс 379
распространение событий 438
спецификация 48
стилевое оформление
вычисленные стили, извлечение 393
именование свойств 391
местонахождение 389
преобразование значений в пикселях 397
указание размеров 398
типы узлов 46
фрагменты, назначение 384
Модули 32
интерфейс, порядок создания 355
назначение 351
определение по стандарту
AMD, особенности 361
CommonJS, особенности 362
ES6, особенности 364, 370
определение, языковые средства 353
привязка по умолчанию 368
расширение 357
требования к модульной системе 353
шаблон
назначение 357
недостатки 360
ограничения 359
экспорт и импорт
именованный и сокращенный синтаксис 367
переименование 369
по умолчанию 368
функциональных возможностей 365

H

Наследование
механизм действия 236
назначение 232
определение 219
реализация с помощью
классов 248
прототипов 219, 233
Настольные приложения 38
Ниндзя 24
Но 24

O

Обещания 32
вызов исполнителя 194
выполнение, порядок 195
неразрешенные, определение 197
определение 172, 194
отклонение, способы 200
применение 202
связывание в цепочку 204
создание, порядок 194

состояния, виды 197
сочетание с генераторами 208
Область видимости
глобальная, назначение 131
определение 130
правила соблюдения 135, 159
связь с лексической средой 143
Обратные вызовы
назначение 66
организация, порядок 69
порядок выполнения 68
трудности
выполнения последовательности шагов 196
обработки ошибок 195
параллельного выполнения ряда шагов 197
Объекты
arguments
как псевдонимы параметров функции 99
назначение 99
document 49
window 49
XMLHttpRequest, назначение 203
выполняемые над ними действия 64
доступ к свойствам через прототипы 220
итераторы, назначение 176
непригодность в качестве отображений 309
операции со свойствами 218
прототипы
присваивание функциям 222
свойство constructor, назначение 223, 230
создание с помощью
ключевого слова class 244
литералов объектов 218
функций-конструкторов и прототипов 222
управление доступом
через методы получения и установки 254
через прокси-объекты 266
Объявления функций
отличия от функциональных выражений 79
применение 77
Операции
instanceof
назначение 240
особенности применения 241
new, назначение 222
typeof, назначение 89
разности, применение 319
расширения, применение 317, 318
сдвига вправо с заполнением нулями, назначение 468
Отладка кода
в JavaScript, методики 488
заход в функцию 491
обход и выход из функции 492
протоколирование 489
современные средства 36, 484
точки останова
по условию, назначение и установка 493
преимущества 490
установка 491
Отображения
обход, порядок 313
определение 306
равенство ключей 312
создание 309

П

Параметры
arguments 95
this 95, 101
Перегрузка верстки
сведение к минимуму 404
явление 403
Перегрузка функций 96
Переменные

закрытые
имитация 160
разъяснение назначения 165
изменяемость 147
объявление, порядок 147
определение в лексических средах 150
типа const
особенности 150
поведение 149
применение 147
частные
имитация 134
Поднятие, назначение 159
Полифиллы
определение 467
реализация, пример 468
Приложения
для мобильных устройств 39
Прокси-объекты 32
определение 266
применение
области 266
примеры 271, 276
суть 269
проблемы с производительностью 278
создание 266
Протоколирование
назначение 270
операторы, назначение 489
применение 489
реализация
без прокси-объектов 270
с помощью прокси-объектов 271
Прототипы
объектов, задание 223
определение 217
связь с получаемыми экземплярами объектов 229
создание, принцип 219
цепочки, образование 221, 233

P

Разработка кросс-браузерного кода
инкапсуляция кода 459
насущные задачи 455
обработка ошибок и отличий в браузерах 456
поглощающие идентификаторы, меры борьбы 461
регрессии, меры борьбы 463
сокращение допущений 472
стратегии реализации
надежное устранения ошибок 465
обнаружение средств и полифиллы 467
Регрессии, определение 463
Регулярные выражения 32
глобальные и локальные совпадения 338
группировка
без фиксации 340
с фиксацией 332
жадные и ленивые операции 330
замена с помощью функций 342
классы символов
операции 328
предопределенные члены 330
компиляция и выполнение 333
модификаторы, разновидности 327
назначение и состав 326
обратные ссылки 332
особенности 323
пассивные подвыражения, определение 341
потенциальные возможности 325
решение типичных задач 344, 346
создание, способы 326
фиксация
назначение 336
обратные ссылки 339
простая 337
члены и операции 328, 332
экранирование специальных символов 329

С

Самурай 24
Свойства
  constructor
    преодоление трудностей переопределения 239
    применение 232
  prototype, назначение 223
  вычисляемые, определение 264
  дескриптор, назначение 237
  объектов
    автоматическое заполнение 274
    выполняемые операции 218
    настройка 237
    проверка достоверности значений 262
    событий, target, назначение 436
    управление доступом 255
    частные, управление доступом 261
  экземпляров, особенности 224
Связывание, сильное и слабое 444
События 34
  асинхронный характер 55
  всплывание, модель обработки 438
  обработка
    главный принцип 56
    делегирование 443
    модели 438
    порядок 53, 436
    по стандарту W3C, порядок 438
  обработчики
    назначение 55
    регистрация, способы 55, 436
  очередь, организация 53
  перехват, модель обработки 438
  специальные
    инициирование 445
    назначение 444
  типы 55
Сортировка 300
Спецификация HTML5, описание 48
Стек
  вызовов, определение 142
  контекстов выполнения
    определение 140
    поведение 140
  назначение и структура 140

Т

Таймеры 34
  методы манипулирования 424
  назначение 424
  применение 433
  тайм-ауты и интервалы времени, отличия 431
Тестирование кода
  качественные тесты, свойства 494
  контрольные примеры
    построение средствами веб-служб 495
  модульное
    в JavaScript, свойства среды 499
    принцип утверждения, реализация 497
  назначение 483
  основные средства 37
  покрытие кода
    измерение 503
    определение 502
  среды тестирования
    Jasmine, назначение и особенности 501
    QUnit, назначение и особенности 500
    назначение 497
    построение 497
  тестовые наборы, организация 499
Типы кода JavaScript
  глобальный код
    выполнение 50
    назначение 49
    определение 139
  код функции
    выполнение 50

назначение 49
определение 139

У

Утверждения 37

Ф

Фиксация 332
Функции 61, 63
alert() 36
assert() 37
применение 67
async 32
define(), применение по стандарту AMD 362
reject(), применение 194
report() 67
resolve(), применение 194
асинхронные, в перспективе 212
в качестве модулей 354
вызов
до объявления 156
как конструктора, особенности 107
как метода, особенности 104
как функции, особенности 103
с помощью методов apply() и call() 115
способы 102
выполняемые над ними действия 65
генераторы
назначение 171
отличие от стандартных функций 191
применение 174
запоминание
достоинства и недостатки 74
определение 73
применение 73
как объекты высшего порядка 63
как основной исполняемый блок 139
конструкторы
возвращаемые значения 110
назначение 107
применение, особенности 112
контекст выполнения, отслеживание 139
немедленно вызываемые, определение 81
новые, создание методом bind() 125
обратного вызова 66
назначение 66
организация, порядок 69
порядок выполнения 68
применение 70
определение, способы 75
параметры и аргументы
назначение псевдонимов параметрам 100
неявные параметры
arguments, назначение 96
this, назначение 101
определение 84
оставшиеся параметры, назначение 87
отличие 84
стандартные параметры, назначение 89
указание 84
свойство prototype, назначение 223
сохранение в коллекции 71
стрелочные
назначение 82
обращение с контекстами функций 120
определение, способы 83
синтаксис 82
Функциональное программирование, особенности 66
Функциональные выражения
немедленно вызываемые
заключение в круглые скобки 81
определение 81
определение 78

отличия от объявлений функций 79
применение 80

Ц

Циклы
for-in, организация 238
for-of, организация и применение 178
while, организация и применение 177
бесконечные, организация в генераторах 180
ожидания событий
выполнение таймеров, особенности 426
макрозадачи и микрозадачи 410
основные принципы 411
очереди микрозадач и макрозадач 411
примеры реализации 414, 419

Я

Язык JavaScript
глобальные объекты, назначение и свойства 49
дальнейшее развитие 32
дополнительные языковые средства 32
замыкания, назначение 130
классы как синтаксическое удобство 245
модель однопоточного выполнения кода, назначение 53
наследование, механизм 236
нормы передовой практики
анализ производительности 37
основные элементы 35
отладка кода 36
тестирование кода 36
обратные вызовы, назначение 66
поддержка в браузерах новых языковых средств 33
разновидности кода JavaScript 49
регулярные выражения, решаемые типичные задачи 323
современное состояние развития 30
строгий режим
назначение 100
отличие от нестрогого режима 104
типы разрабатываемых приложений 38
транспиляторы, применение 33
функционально ориентированный, отличительные особенности 31

Шаблонные литералы позволяют встраивать выражения в символьные строки: `` `${ninja}` ``.

Переменные с блочной областью видимости:

- Пользуйтесь новым ключевым словом `let` для создания переменных с блочной областью видимости: `let ninja = "Yoshi"`.
- Пользуйтесь новым ключевым словом `const` для создания таких переменных с блочной областью видимости, которым нельзя повторно присвоить новое значение: `const ninja = "Yoshi"`.

Параметры функций

- *Оставшиеся параметры* образуют массив из тех аргументов, которые не соответствуют параметрам функции:

```
function multiMax(first, ...remaining){
/*...*/
}
multiMax(2, 3, 4, 5);
//first: 2; remaining: [3, 4, 5]
```

- *Стандартные параметры* определяют значения, используемые по умолчанию, если при вызове функции не указаны конкретные значения:

```
function do(ninja, action = "skulk"){
return ninja + " " + action;
}
do("Fuma"); // возвращается строка "Fuma skulk"
```

Операции расширения позволяют расширить выражение там, где требуется несколько элементов: `[...items, 3, 4, 5]`.

Стрелочные функции позволяют создавать менее громоздкие в синтаксическом плане функции. У них отсутствует собственный параметр `this`. Вместо этого они наследуют его из того контекста, где были созданы:

```
const values = [0, 3, 2, 5, 7, 4, 8, 1];
values.sort((v1,v2) => v1 - v2);
/*ИЛИ*/
values.sort((v1,v2) => { return v1 - v2;});
value.forEach(value => console.log(value));
```

Генераторы формируют последовательности значений на основе запросов. Как только значение будет сформировано, генератор прервет свое выполнение без блокировки основной программы. Пользуйтесь ключевым словом `yield` для формирования значений:

```
function *IdGenerator(){
    let id = 0;
    while(true){
```

```
      yield ++id;
    }
  }
```

Обещания (обязательства) служат заполнителями результатов вычислений. Обещание гарантирует, что результат вычисления в конечном итоге станет известен. Обещание может быть либо выполнено, либо отклонено, после чего никакие изменения больше невозможны:

- Создавайте новое обещание с помощью операции `new Promise((resolve, reject) => {});`.
- Вызывайте функцию `resolve()`, чтобы разрешить обещание явным образом. Вызывайте функцию `reject()`, чтобы отклонить обещание явным образом.
  Если возникает ошибка, обещание отклоняется неявным образом.
- У объекта обещания имеется метод `then()`, возвращающий обещание, которому передаются две функции обратного вызова, вызываемые при удачном и неудачном исходе:
  ```
  myPromise.then(val => console.log("Success"),
                 err => console.log("Error"));
  ```
- Связывайте методы `catch()` в цепочку, чтобы перехватывать отклоненные обещания:
  ```
  myPromise.catch(e => alert(e));
  ```

Классы служат для удобства синтаксического представления прототипов в JavaScript:.

```
class Person {
  constructor(name){ this.name = name; }
  dance(){ return true; }
}

class Ninja extends Person {
  constructor(name, level){
    super(name);
    this.level = level;
  }
  static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
  }
}
```

Прокси-объекты управляют доступом к другим объектам. Для взаимодействия с объектом требуются специальные действия (например, когда читается значение свойства или вызывается функция):

```
const p = new Proxy(target, {
  get: (target, key) => {
    /* Вызывается при доступе к свойству через
       прокси-объект */ },
```

```
  set: (target, key, value) => {
    /* Вызывается при установке свойства через
       прокси-объект */ }
});
```

Отображения определяют взаимно-однозначное соответствие ключей и значений:

- Операция `new Map()` создает новое отображение.
- Пользуйтесь методом `set()` для добавления нового отображения, методом `get()` — для извлечения существующего отображения, методом `has()` — для проверки факта существования отображения, а методом `delete()` — для удаления отображения.

Множества являются коллекциями однозначных элементов:

- Операция `new Set()` создает новое множество.
- Пользуйтесь методом `add()` для добавления нового элемента в множество, методом `delete()` — для удаления элемента из множества, а свойством `size` — для проверки количества элементов в множестве.

Циклы `for-of` служат для перебора коллекций и генераторов.

При *деструктурировании* данные извлекаются из объектов и массивов:

- `const {name: ninjaName} = ninja;`
- `const [firstNinja] = ["Yoshi"];`

Модули служат более крупными единицами организации кода, позволяющими разделять программы на части:

```
export class Ninja{}; // экспортировать элемент
export default class Ninja{} // экспортировать класс по
                             // умолчанию
export {ninja};// экспортировать существующие переменные
export {ninja as samurai}; // переименовать экспорт

import Ninja from "Ninja.js"; // импортировать экспорт
                              // по умолчанию
import {ninja} from "Ninja.js"; // импортировать именованные
                                // виды экспорта
import * as Ninja from "Ninja.js"; // импортировать все
                                   // именованные виды
                                   // экспорта
import {ninja as iNinja} from "Ninja.js"; // импортировать
                                          // с новым именем
```